推酷

编程狂人
Programming Madman

NO.45

关于推酷

推酷是专注于IT圈的个性化阅读社区。我们利用智能算法, 从海量文章资讯中挖掘出高质量的内容,并通过分析用户 的阅读偏好,准实时推荐给你最感兴趣的内容。我们推荐 的内容包含科技、创业、设计、技术、营销等内容,满足 你日常的专业阅读需要。我们针对IT人还做了个活动频道, 它聚合了IT圈最新最全的线上线下活动,使IT人能更方便地 找到感兴趣的活动信息。

关于周刊

《编程狂人》是献给广大程序员们的技术周刊。我们利用技术挖掘出那些高质量的文章,并通过人工加以筛选出来。每期的周刊一般会在周二的某个时间点发布,敬请关注阅读。

本期为精简版 周刊完整版链接:http://www.tuicool.com/mags/543bdf95d91b1446420889cd

欢迎下载推酷客户端体验更多阅读乐趣

目录

互联网全站HTTPS的时代已经到来

作者：BAIDU罗成

前言

我目前在百度从事HTTPS方面的性能优化工作。百度无线搜索目前已经支持https，手机访问地址是https://m.baidu.com。

在HTTPS项目的开展过程中明显感觉到目前国内互联网对HTTPS并不是很重视，其实也就是对用户隐私和网络安全不重视。本文从保护用户隐私的角度出 发，简单描述现在存在的用户隐私泄露和流量劫持现象，然后进一步说明为什么HTTPS能够保护用户安全以及HTTPS使用过程中需要注意的地方。

国外很多网站包括google,facebook,twitter都支持了全站HTTPS，而国内目前还没有一家大型网站全站支持HTTPS（PC端的 微信全部使用了HTTPS，但是PC端用户应该不多）。甚至一些大型网站明显存在很多HTTPS使用不规范或者过时的地方。比如支付宝使用的是 tls1.0和RC4,而京东(quickpay.jd.com)竟然还使用着SSL3.0这个早就不安全并且性能低下的协议，其他很多网站的HTTPS 登陆页面也存在着不安全的HTTP链接，这个也为黑客提供了可乘之机。

由于篇幅关系，文中几乎没有详细描述任何细节，后面有时间我再一一整理成博客发表出来。同时由于水平有限，本文肯定存在很多错误，希望大家不吝赐教。本 文的大部分内容都能从互联网上搜索到，有些地方我也标明了引用，可以直接跳转过去。但是全文都是在我自己理解的基础上结合开发部署过程中的一些经验和测试 数据一个字一个字敲出来的，最后决定将它们分享出来的原因是希望能和大家多多交流，抛砖引玉，共同推进中国互联网的HTTPS发展。

本文不会科普介绍HTTPS、TLS及PKI，如果遇到一些基本概念文中只是提及而没有描述，请大家自行百度和google。本文重点是想告诉大家HTTPS没有想像中难用可怕，只是没有经过优化。

中国互联网全站使用HTTPS的时代已经到来。

1，用户隐私泄露的风险很大

人们的生活现在已经越来越离不开互联网，不管是社交、购物还是搜索，互联网都能带给人们很多的便捷。与此同时，用户“裸露”在互联网的信息也越来越多，另一个问题也日益严重，那就是隐私和安全。

几乎所有的互联网公司都存在用户隐私泄露和流量劫持的风险。BAT树大招风，这方面的问题尤其严重。比如用户在百度搜索一个关键词，“人流”，很快就会有 医院打电话过来推销人流手术广告，不知情的用户还以为是百度出卖了他的手机号和搜索信息。同样地，用户在淘宝搜索的关键词也很容易被第三方截获并私下通过 电话或者其他广告形式骚扰用户。而QQ和微信呢，显然用户不希望自己的聊天内容被其他人轻易知道。为什么BAT不可能出卖用户隐私信息给第三方呢？因为保 护用户隐私是任何一个想要长期发展的互联网公司的安身立命之本，如果用户发现使用一个公司的产品存在严重的隐私泄露问题，显然不会再信任该公司的产品，最 终该公司也会因为用户大量流失而陷入危机。所以任何一家大型互联网公司都不可能因为短期利益而出卖甚至忽视用户隐私。

那既然互联网公司都知道用户隐私的重要性，是不是用户隐私就得到了很好的保护呢？现实却并不尽如人意。由于目前的WEB应用和网站绝大部分是基于 HTTP协议，国内没有任何一家大型互联网公司采用全站HTTPS方案来保护用户隐私（排除支付和登陆相关的网站或者页面以及PC端的微信）。因为 HTTP协议简单方便，易于部署，并且设计之初也没有考虑安全性，所有内容都是明文传输，也就为现在的安全问题埋下了隐患。用户在基于HTTP协议的 WEB应用上的传输内容都可以被中间者轻易查看和修改。

比如你在百度搜索了一个关键词“https“，中间者通过tcpdump或者wireshark等工具就很容易知道发送请求的全部内容。wireshark的截图如下：

Stream Content

```
GET / HTTP/1.1
Host: www.baidu.com
User-Agent: Mozilla/5.0 (Windows NT 6.3;
Accept: text/html,application/xhtml+xml,a
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Cookie: BAIDUID=ED3E6D5556AEC17B7AD052668
BDUSS=E0c0RHNGlMaU9BVnNTZGI2RXlCcjF3LTE1c
ZVc0WTdhSnM1U0dZdlZUQVFBQUFBJCQAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Connection: keep-alive

HTTP/1.1 200 OK
Date: Sat, 04 Oct 2014 17:46:07 GMT
Content-Type: text/html
Transfer-Encoding: chunked
Connection: Keep-Alive
Cache-Control: private
Expires: Sat, 04 Oct 2014 17:46:07 GMT
Content-Encoding: gzip
Server: BWS/1.1
BDPAGETYPE: 2
BDQID: 0xd89b2e8f0078805a
BDUSERID: 264493947
Set-Cookie: BDSVRTM=102; path=/
Set-Cookie: BD_HOME=1; path=/
```


这里所谓的中间者是指网络传输内容需要经过的网络节点，既有硬件也有软件，比如中间代理服务器、路由器、小区WIFI热点、公司统一网关出口等。这 里面最容易拿到用户内容的就是各种通信服务运营商和二级网络带宽提供商。而最有可能被第三方黑客动手脚的就是离用户相对较近的节点。

中间者为什么要查看或者修改用户真实请求内容呢？很简单，为了利益。常见的几种危害比较大的中间内容劫持形式如下：

1. 获取无线用户的手机号和搜索内容并私下通过电话广告骚扰用户。为什么能够获取用户手机号？呵呵，因为跟运营商有合作。

2. 获取用户帐号cookie，盗取帐号有用信息。

3. 在用户目的网站返回的内容里添加第三方内容，比如广告、钓鱼链接、植入木马等。

总结来讲，由于HTTP明文传输，同时中间内容劫持的利益巨大，所以用户隐私泄露的风险非常高。

2，HTTPS能有效保护用户隐私

HTTPS就等于HTTP加上TLS（SSL)，HTTPS协议的目标主要有三个：

1. 数据保密性。保证内容在传输过程中不会被第三方查看到。就像快递员传递包裹时都进行了封装，别人无法知道里面装了什么东西。

2. 数据完整性。及时发现被第三方篡改的传输内容。就像快递员虽然不知道包裹里装了什么东西，但他有可能中途掉包，数据完整性就是指如果被掉包，我们能轻松发现并拒收。

3. 身份校验。保证数据到达用户期望的目的地。就像我们邮寄包裹时，虽然是一个封装好的未掉包的包裹，但必须确定这个包裹不会送错地方。

通俗地描述上述三个目标就是封装加密，防篡改掉包，防止身份冒充，那TLS是如何做到上述三点的呢？我分别简述一下。

2.1 数据保密性

2.1.1 非对称加密及密钥交换

数据的保密性主要是通过加密完成的。加密算法一般分为两种，一种是非对称加密（也叫公钥加密），另外一种是对称加密（也叫密钥加密）。所谓非对称加密就是指加密和解密使用的密钥不一样，如下图：

HTTPS使用非对称加解密主要有两个作用，一个是密钥协商，另外可以用来做数字签名。所谓密钥协商简单说就是根据双方各自的信息计算得出双方传输内容时对称加解密需要使用的密钥。

公钥加密过程一般都是服务器掌握私钥，客户端掌握公钥，私钥用来解密，公钥用来加密。公钥可以发放给任何人知道，但是私钥只有服务器掌握，所以公钥加解 密非常安全。当然这个安全性必须建立在公钥长度足够大的基础上，目前公钥最低安全长度也需要达到2048位。大的CA也不再支持2048位以下的企业级证 书申请。因为1024位及以下的公钥长度已经不再安全，可以被高性能计算机比如量子计算机强行破解。计算性能基本会随着公钥的长度而呈2的指数级下降。

既然如此为什么还需要对称加密？为什么不一直使用非对称加密算法来完成全部的加解密过程？主要是两点：

1. 非对称加解密对性能的消耗非常大，一次完全TLS握手，密钥交换时的非对称解密计算量占整个握手过程的95%。而对称加密的计算量只相当于非对称加密的0.1%，如果应用层数据也使用非对称加解密，性能开销太大，无法承受。

2. 非对称加密算法对加密内容的长度有限制，不能超过公钥长度。比如现在常用的公钥长度是2048位，意味着待加密内容不能超过256个字节。

目前常用的非对称加密算法是RSA，想强调一点的就是RSA是整个PKI体系及加解密领域里最重要的算法。如果想深入理解HTTPS的各个方面，RSA是必需要掌握的知识点。它的原理主要依赖于三点：

1. 乘法的不可逆特性。即我们很容易由两个乘数求出它们的积，但是给定一个乘积，很难求出它是由哪两个乘数因子相乘得出的。

2. 欧拉函数。欧拉函数是小于或等于n的正整数中与n互质的数的数目

3. 费马小定理。假如a是一个整数，p是一个质数，那么是p的倍数。

这篇中文博客对RSA的原理解释得比较清楚易懂：RSA算法原理。

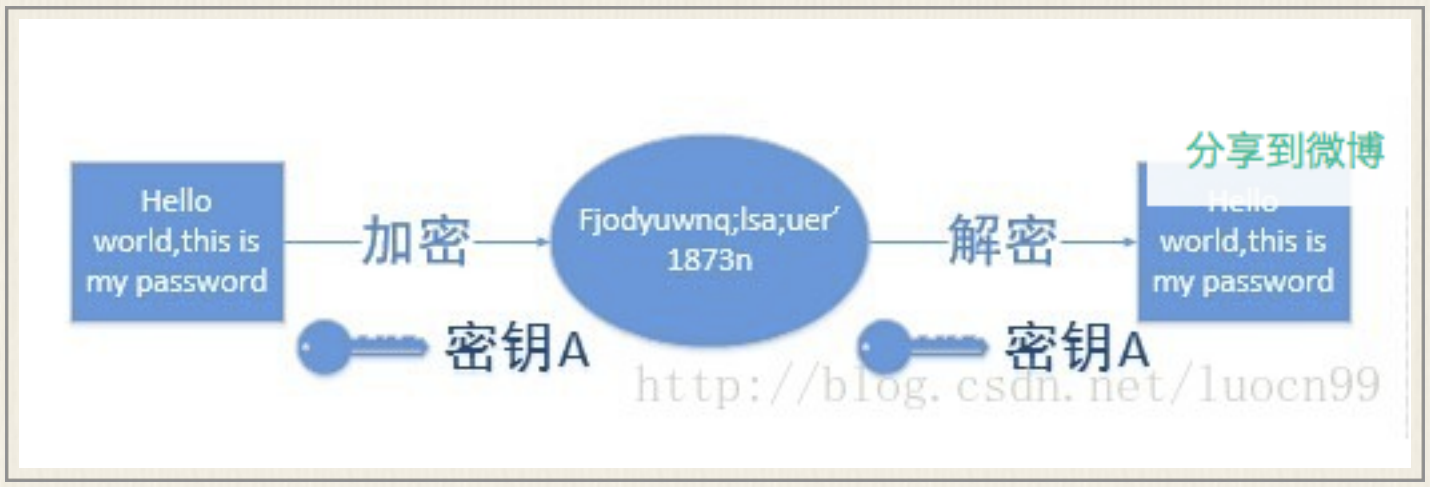
RSA算法是第一个也是目前唯一一个既能用于密钥交换又能用于数字签名的算法。另外一个非常重要的密钥协商算法是diffie- hellman(DH).DH不

需要预先知道通信双方的信息就能完成密钥的协商，它使用一个素数P的整数乘法群以及原根G，理论依据就是离散对数。

openssl目前只支持如下密钥交换算法：RSA，DH，ECDH, DHE，ECDHE。各个算法的性能和对速度的影响可以参考后面章节，由于篇幅有限，具体实现不再做详细介绍。

2.1.2 对称加密

对称加密就是加密和解密都使用的是同一个密钥。如下图：



采用非对称密码算法的密钥协商过程结束之后就已经得出了本次会话需要使用的对称密钥。对称加密又分为两种模式：流式加密和分组加密。流式加密现在常用的就是RC4，不过RC4已经不再安全，微软也建议网站尽量不要使用RC4流式加密。支付宝可能没有意识到这一点，也可能是由于其他原因，他们仍然在使用RC4算法和TLS1.0协议。

一种新的替代RC4的流式加密算法叫ChaCha20，它是google推出的速度更快，更安全的加密算法。目前已经被android和chrome采用，也编译进了google的开源openssl分支---boring ssl，并且nginx 1.7.4也支持编译boringssl。我目前还没有比较这种算法的性能，但部分资料显示这个算法对性能的消耗比较小，特别是移动端提升比较明显。

分组加密以前常用的模式是AES-CBC，但是CBC已经被证明容易遭受BEAST和LUCKY13攻击。目前建议使用的分组加密模式是AES-GCM，不过它的缺点是计算量大，性能和电量消耗都比较高，不适用于移动电话和平板电脑。尽管如此，它仍然是我们的优先选择。

2.2 数据完整性

这 部分内容相对比较简单，openssl现在使用的完整性校验算法有两种：MD5或者SHA。由于MD5在实际应用中存在冲突的可能性比较大，所以尽量别采 用MD5来验证内容一致性。SHA也不能使用SHA0和SHA 1 ，中国山东大学的王小云教授在2005年就牛逼地宣布破解了SHA-1完整版算法。建议使 用SHA 2 算法，即输出的摘要长度超过224位。

2.3 身份验证和授权

这里主要介绍的就是PKI和数字证书。数字证书有两个作用：

1. 身份验证。确保客户端访问的网站是经过CA验证的可信任的网站。

2. 分发公钥。每个数字证书都包含了注册者生成的公钥。在SSL握手时会通过certificate消息传输给客户端。

这里简单介绍一下数字证书是如何验证网站身份的。

证书申请者首先会生成一对密钥，包含公钥和密钥，然后把公钥及域名还有CU等资料制作成CSR格式的请求发送给RA，RA验证完这些内容之后（RA会请独立的第三方机构和律师团队确认申请者的身份）再将CSR发送给CA，CA然后制作X.509格式的证书。

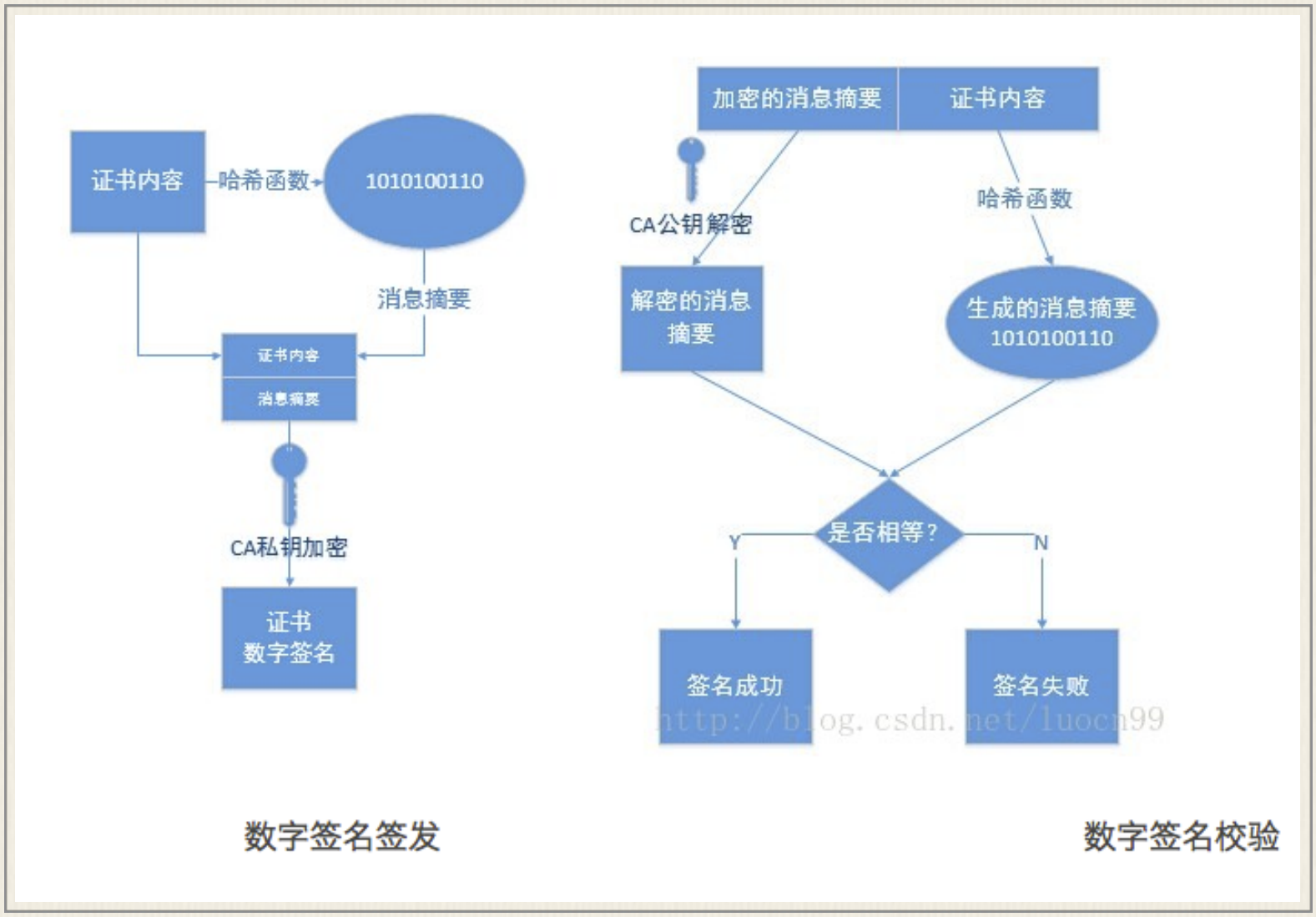
那好，申请者拿到CA的证书并部署在网站服务器端，那浏览器访问时接收到证书后，如何确认这个证书就是CA签发的呢？怎样避免第三方伪造这个证书?

答案就是数字签名（digital signature）。数字签名可以认为是一个证书的防伪标签，目前使用最广泛的SHA-
RSA数字签名的制作和验证过程如下：

1. 数字签名的签发。首先是使用哈希函数对证书数据哈希，生成消息摘要，然后使用CA自己的私钥对证书内容和消息摘要进行加密。

2. 数字签名的校验。使用CA的公钥解密签名，然后使用相同的签名函数对证书内容进行签名并和服务端的数字签名里的签名内容进行比较，如果相同就认为校验成功。

图形表示如下：



这里有几点需要说明：

1. 数字签名签发和校验使用的密钥对是CA自己的公私密钥，跟证书申请者提交的公钥没有关系。

2. 数字签名的签发过程跟公钥加密的过程刚好相反，即是用私钥加密，公钥解密。

3. 现在大的CA都会有证书链，证书链的好处一是安全，保持根CA的私钥离线使用。第二个好处是方便部署和撤销，即如何证书出现问题，只需要撤销相应级别的证书，根证书依然安全。

4. 根CA证书都是自签名，即用自己的公钥和私钥完成了签名的制作和验证。而证书链上的证书签名都是使用上一级证书的密钥对完成签名和验证的。

5. 怎样获取根CA和多级CA的密钥对？它们是否可信？当然可信，因为这些厂商跟浏览器和操作系统都有合作，它们的公钥都默认装到了浏览器或者操作系统环境里。比如firefox就自己维护了一个可信任的CA列表，而chrome和IE使用的是操作系统的CA列表。

数字证书的费用其实也不高，对于中小网站可以使用便宜甚至免费的数字证书服务（可能存在安全隐患），像著名的verisign公司的证书一般也就几千到几万块一年不等。当然如果公司对证书的需求比较大，定制性要求高，可以建立自己的CA站点，比如google，能够随意签发google相关证书。

3，HTTPS对速度和性能的影响

既然HTTPS非常安全，数字证书费用也不高，那为什么互联网公司不全部使用HTTPS呢？原因主要有两点：

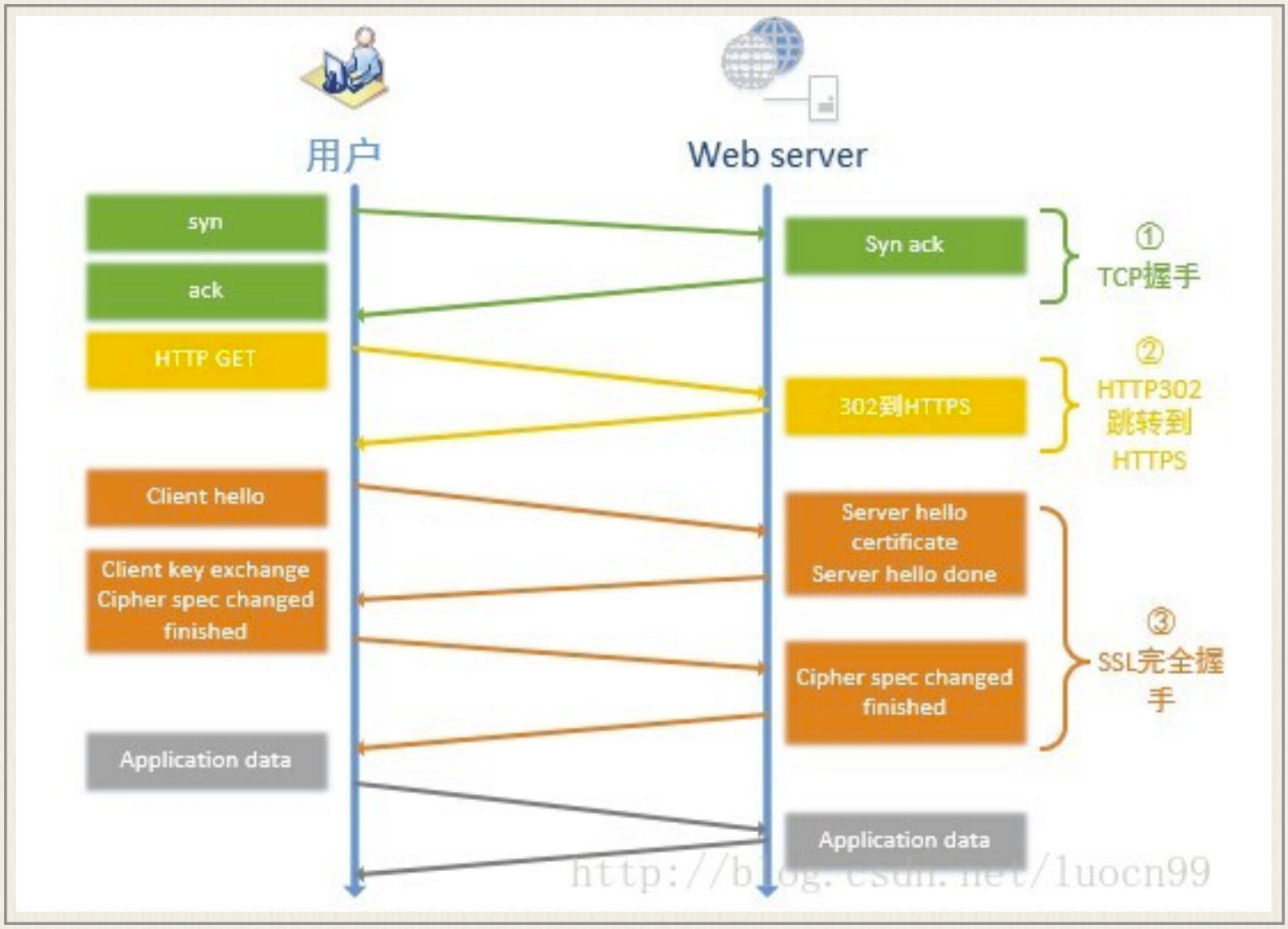
1. HTTPS对速度的影响非常明显。每个HTTPS连接一般会增加1-3个RTT，加上加解密对性能的消耗，延时还有可能再增加几十毫秒。

2. HTTPS对CPU计算能力的消耗很严重，完全握手时，web server的处理能力会降低至HTTP的10%甚至以下。

下面简单分析一下这两点。

3.1 HTTPS对访问速度的影响

我用一张图来表示一个用户访问使用HTTPS网站可能增加的延时：



HTTPS增加的延时主要体现在三个阶段，包含了上图所示的2和3阶段。

1. 302跳转。为什么需要302？因为用户懒。我想绝大部分网民平时访问百度时都是输入www.baidu.com或者baidu.com吧？很少有输入http://www.baidu.com访问百度搜索的吧？至于直接输入https://www.baidu.com来访问百度的HTTPS服务的就更加少了。所以为了强制用户使用HTTPS服务，只有将用户发起的HTTP请求www.baidu.com302成https://www.baidu.com。这无疑是增加一个RTT的跳转延时。

2. 上图第三阶段的SSL完全握手对延时的影响就更加明显了，这个影响不仅体现在网络传输的RTT上，还包含了数字签名的校验，由于客户端特别是移动端的计算性能弱，增加几十毫秒的计算延时是很常见的。

3. 还 有一个延时没有画出来，就是证书的状态检查，现在稍微新一点的浏览器都使用ocsp来检查证书的撤销状态，在拿到服务器的证书内容之后会访问ocsp站点 获取证书的状态，检查证书是否撤销。如果这个ocsp站点在国外或者ocsp服务器出现故障，显然会影响这个正常用户的访问速

度。不过还好ocsp的检查 周期一般都是7天一次，所以这个对速度的影响还不是很频繁。 另外chrome默认是关闭了ocsp及crl功能，最新版的firefox开启了这个功能，如果ocsp返回不正确，用户无法打开访问网站。

实际测试发现，在没有任何优化的情况下，HTTPS会增加200ms以上的延时。

那是不是对于这些延时我们就无法优化了呢？显然不是，部分优化方式参考如下：

1. 服务器端配置HSTS，减少302跳转，其实HSTS的最大作用是防止302 HTTP劫持。HSTS的缺点是浏览器支持率不高，另外配置HSTS后HTTPS很难实时降级成HTTP。

2. 设置ssl session 的共享内存cache. 以nginx为例，它目前只支持session cache的单机多进程共享。配置如下：

```
ssl_session_cache    shared:SSL:10m;
```

如 果是前端接入是多服务器架构，这样的session cache是没有作用的，所以需要实现session cache的多机共享机制。我们已经在nginx 1.6.0版本上实现了多机共享的session cache。多机session cache的问题必须要同步访问外部session cache，比如redis。由于openssl目前提供的API是同步的，所以我们正在改进openssl和nginx的异步实现。

3. 配 置相同的session ticket key，部署在多个服务器上，这样多个不同的服务器也能产生相同的 session ticket。session ticket 的缺点是支持率不广，大概只有40%。而session id是client hello的标准内容，从SSL2.0开始就被全部客户支持。

```
ssl_session_tickets    on;
ssl_session_ticket_key ticket_keys;
```

4. 设置ocsp stapling file，这样ocsp的请求就不会发往ca提供的ocsp站点，而是发往网站的webserver。ocsp的配置和生成命令如下：

ssl_stapling on;

ssl_stapling_file domain.staple;

上面是*nginx*配置，如下是*ocsp_stapling_file*的生成命令：

openssl s_client -showcerts -connect yourdomain:443 < /dev/null | awk -v c=-1 '/-----BEGIN CERTIFICATE-----/{inc=1;c++} inc {print > ("level" c ".crt")} /---END CERTIFICATE-----/{inc=0}'

for i in level?.crt;

do

openssl x509 -noout -serial -subject -issuer -in "$i";

echo;

done

openssl ocsp -text -no_nonce -issuer level1.crt -CAfile CAbundle.crt -cert level0.crt -VAfile level1.crt -url $ocsp_url -respout domain.staple , 其中*$ocsp_url*等于*ocsp*站点的*URL*，可以通过如下命令求出：
for i in level?.crt; do echo "$i:"; openssl x509 -noout -text -in "$i" | grep OCSP; done，如果是证书链，一般是最底层的值。

5. 优先使用ecdhe密钥交换算法，因为它支持PFS(perfect forward secrecy)，能够实现false start。

6. 设置tls record size，最好是能动态调整record size，即连接刚建立时record size设置成msg，连接稳定之后可以将record size动态增加。

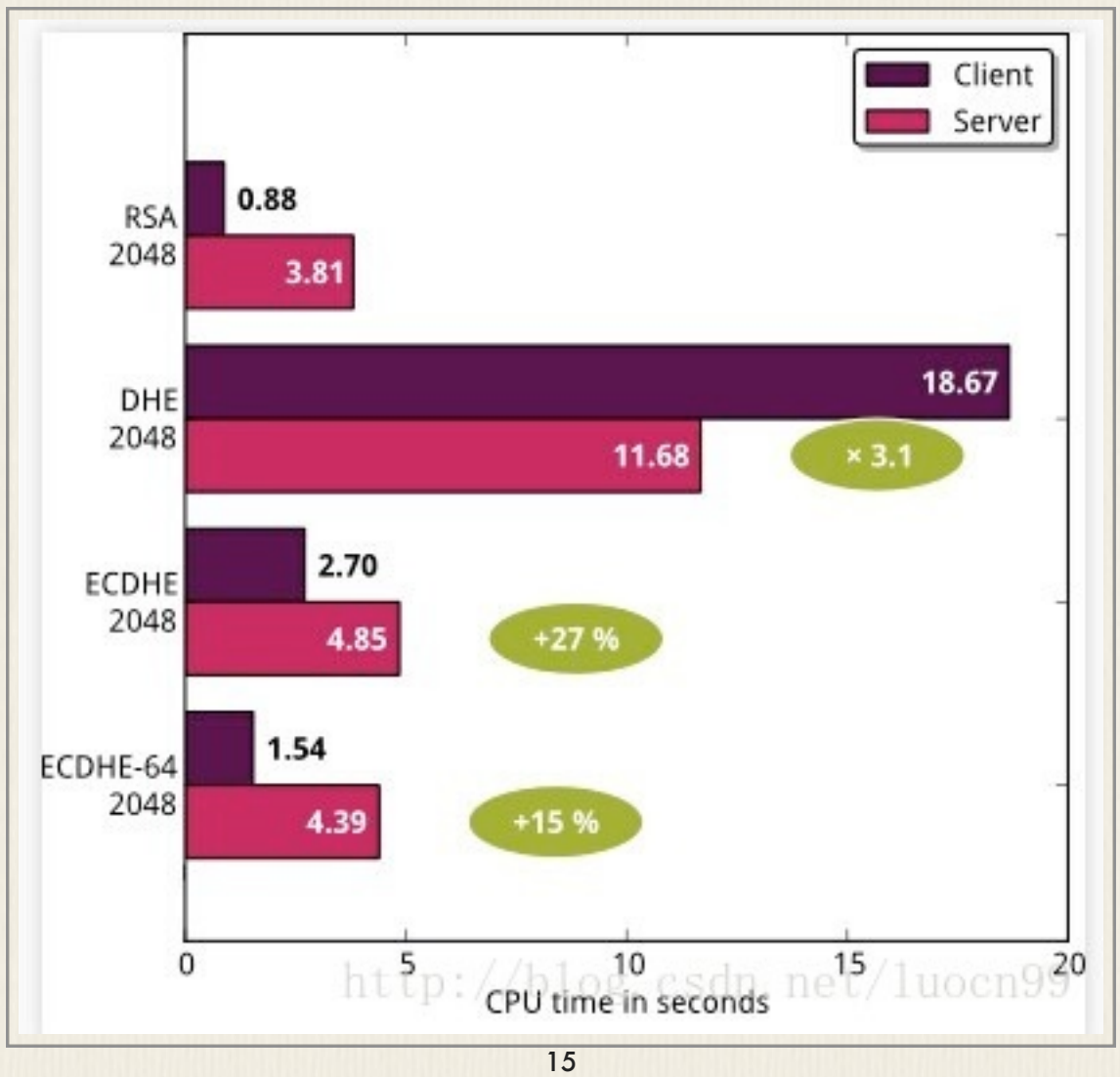
7. 如果有条件的话可以启用tcp fast open。虽然现在没有什么客户端支持。

8. 启 用SPDY。SPDY是强制使用HTTPS的，协议比较复杂，需要单独的文章来分析。可以肯定的一点是使用SPDY的请求不仅明显提升了HTTPS速度， 甚至比HTTP还要快。在无线WIFI环境下，SPDY比HTTP要快50ms左右，3G环境下比HTTP要快250ms。

3.2 HTTPS 对性能的影响

HTTPS为什么会严重降低性能？主要是握手阶段时的大数运算。其中最消耗性能的又是密钥交换时的私钥解密阶段（函数是rsa_private_decryption）。这个阶段的性能消耗占整个SSL握手性能消耗的95%。

前面提及了openssl密钥交换使用的算法只有四种：rsa, dhe, ecdhe，dh。dh由于安全问题目前使用得非常少，所以这里可以比较下前面三种密钥交换算法的性能，具体的数据如下：



上图数据是指完成1000次握手需要的时间，显然时间数值越大表示性能越低。图片和测试方法都可以参考原文，地址如下：http://vincent.bernat.im/en/blog/2011-ssl-perfect-forward-secrecy.html。

密钥交换步骤是SSL完全握手过程中无法绕过的一个阶段。我们只能采取如下措施：

1. 通过session cache和session ticket提升session re-use率，减少完全握手(full handshake)次数，提升简化握手(abbreviated handshake)率。

2. 出于前向加密和false start的考虑，我们优先配置ecdhe用于密钥交换，但是性能不足的情况下可以将rsa配置成密钥交换算法，提升性能。

openssl 自带的工具可以计算出对称加密、数字签名及HASH函数的各个性能，所以详细数据我就不再列举，读者可以自行测试 。

结论就是对称加密RC4的性能最快，但是RC4本身不安全，所以还是正常情况下还是采用AES。HASH函数MD5和SHA1差不多。数字签名是ecdsa算法最快，但是支持率不高。

事实上由于密钥交换在整个握手过程中消耗性能占了95%，而对称加解密的性能消耗不到0.1%，所以server端对称加密的优化收益不大。相反，由于客户端特别是移动端的CPU计算能力本来就比较弱，所以对称加密和数字签名的优化主要是针对移动客户端。

poly1350是google推出的号称优于aes-gcm的对称加密算法，适用于移动端，可以试用一下。

最后经过测试，综合安全和性能的最优cipher suite配置是：ECDHE-RSA-AES128-GCM-SHA256.

如果性能出现大幅度下降，可以修改配置，提升性能但是弱化了安全性，配置是：rc4-md5，根据openssl的规则，密钥交换和数字签名默认都是使用rsa。

4，HTTPS的支持率分析

分析了百度服务器端一百万的无线访问日志（主要为手机和平板电脑的浏览器），得出协议和握手时间的关系如下：

tls协议版本	客户端使用率	握手时间ms
tls 1.2	24.8%	299.496
tls 1.1	0.9%	279.383
tls 1.0	74%	307.077
ssl 3.0	0.3%	484.564

从上表可以发现，ssl3.0速度最慢，不过支持率非常低。tls 1.0支持率最广泛。加密套件和握手时间的关系如下：

加密套件	客户端使用率	握手时间
ECDHE-RSA-AES128-SHA	58.5%	294.36
ECDHE-RSA-AES128-SHA256	21.1%	303.065
DHE-RSA-AES128-SHA	16.7%	351.063
ECDHE-RSA-AES128-GCM-SHA256	3.7%	274.83

显然DHE对速度的影响比较大，ECDHE的性能确实要好出很多，而AES128-GCM对速度也有一点提升。

通过tcpdump分析client hello请求，发现有56.53%的请求发送了session id。也就意味着这些请求都能通过session cache得到复用。其他的一些扩展属性支持率如下：

tls扩展名	支持率
server_name	76.99%
session_tickets	38.6%
next_protocol_negotiation	40.54%
elliptic_curves	90.6%
ec_point_formats	90.6%

这几个扩展都非常有意义，解释如下：

server_name,，即 sni （server name indicator)，有77%的请求会在client hello里面携带想要访问的域名，允许服务端使用一个IP支持多个域名。

next_protocol_negotiation，即NPN，意味着有40.54%的客户端支持spdy.

session_tickets只有38.6%的支持率，比较低。这也是我们为什么会修改nginx主干代码实现session cache多机共享机制的原因。

elliptic_curves即是之前介绍的ECC（椭圆曲线系列算法），能够使用更小KEY长度实现DH同样级别的安全，极大提升运算性能。

5，结论

现在互联网上HTTPS的中文资料相对较少，同时由于HTTPS涉及到大量协议、密码学及PKI体系的知识，学习门槛相对较高。另外在具体的实

践过程中 还有很多坑和待持续改进的地方。希望本文对大家有一些帮助，同时由于我本人在很多地方掌握得也比较粗浅，一知半解，希望大家能多提意见，共同进步。

最后，为了防止流量劫持，保护用户隐私，大家都使用HTTPS吧，全网站支持。事实上，HTTPS并没有那么难用和可怕，只是你没有好好优化。

原文链接：http://blog.csdn.net/luocn99/article/details/39777707

如何选择Node.js Web开发框架?

作者：闲散人生

Node.js非常适用于Web开发，但是现在无论是一个网站，还是Web App都已经成为包括很多不同部分，如前端、数据库、业务模块、功能模块等等的大型项目，使用Node.js从零开始进行Web开发，也许大中型团队能够 胜任，但对于个人和小型团队来说是不现实的。这时候框架就成为Web开发利器，对于个人开发来说几乎是必不可少。那么如何选择Node.js Web开发框架呢?

首先，我们必须要弄清楚的是，我们需要的是——\

程序 or 框架?

程序是已经成型的应用，你需要的是为它搭建环境、添加配置，然后就可以运行起来；框架则是应用的骨架，你需要为它添加数据模型、业务逻辑，它才能成为应用，开始提供服务。

事实上，对于Web开发来说，程序和框架的区别正越来越模糊，比如几乎妇孺皆知的Wordpress，它是一个博客程序，但它丰富的插件以及高度的自定义能够支持很大程度上的二次开发，在这点上它比起一些PHP框架也并不逊色。我个人认为，如果重心在于提供服务而不是掌握技术，有WordPress 这样的程序是没有必要使用框架的。

可惜的是，由于Nodejs还很年轻，目前还没有WordPress这样的程序，因此目前在Node.js开发里，如果想做出自己想要的作品，框架是必然的选择。如果是某些特定类型的应用，可以尝试一些开源的程序，比如要用Nodejs做博客，有Hexo、Ghost等。

Node.js Web框架有哪些?

Node.js里的Web框架分为API框架和Web应用框架。前者能够开发出RESTful的API，后者也能开发出RESTful API，但还包括模板、渲染等为前端所准备的功能。

API框架的使用场景是为跨平台应用提供统一的数据模型，而渲染由前端/客户端自行解决。目前比较知名的API框架有

- restify
- ActionHero.js
- LoopBack
- Fortune.js
- Frisby （经提醒修正，这是一个用于测试RESTful API的框架，并不是API框架。）

Web应用框架顾名思义，就是为了打造Web应用所开发的框架。这里有两种风格的Web应用框架。

一个是Sinatra风格，另一个是Rails风格。Sinatra和Rails都是Ruby语言的Web框架，后者的影响力更大也更为知名。这里简单的解释一下两种风格是什么意思。

Sinatra风格是指高度可配置，注重开发的自由度。代表性的Nodejs Web框架有:

- Express（TJ大神开发，Node.js官方推荐）
- hapi
- koa.js
- flaliron
- total.js
- locomotive

Rails风格则是指不重复自己和约定优于配置，以及严格遵循MVC结构开发。代表性的框架有:

- Sails.js
- geddy
- CompoundJS（原railswayjs）

这两种风格无所谓谁优谁劣，全凭使用者的偏好。

而在这两种Web框架之外，还有更大型的框架，即全栈框架，其中的代表是MEAN。

MEAN?

MEAN指MongoDB+Express+Angular.js+Node.js，这一组合包括运行环境、数据库、Web框架和前端引擎。被称为 全栈框架（Full-stack framework）。这其中除了Node.js之外，每一个都是可替换的，目标是创建从前端到后端，全部使用javascript的Web应用。

由于这一框架的完善性，有人将其称为LAMP的接班人。LAMP即PHP的典型运行环境，Linux+Apache+MySql+PHP，被大量的用于各种虚拟主机上。

MEAN看似庞大，但事实上要构建完整的现代化Web应用，特别是SPA（单页面应用），这几个组件都是难以缺少的，并且，其中每一项几乎都是目前 情况下的最佳选择，因此用于学习和重头开始打造新的Web应用是非常合适的。但由于实际业务的独特性，很可能要替换其中的组件，比如用Mysql来替换 MongoD-
B，因此，学习其中的原理和架构，打造自己的类MEAN框架也是一种选择。

作为个人和小团队来说，全栈框架MEAN基本上足够了，但目前大多数全栈框架还包含一项特性，那就是实时，拥有实时功能的框架我们又称为实时框架。

实时框架好吗?

实时框架（Real-time framework）指包含了webSocket的双向通信功能，能够在服务器和客户端做到实时通信的框架。

服务端和客户端自由通信的需求一直都在，但由于HTTP协议本身的局限性，因此催生了Comet等变通的方法，但即使这样也离实时相距甚远。而当Node.js兴起后，另一个HTML5技术webSocket也渐渐成熟，人们突然发现，实时通信一下子变得触手可及，于是webSocket技术在 Node.js中得到大量的应用，其中最为知名的模块就是socket.io，而各种全栈框架也纷纷加入实时特性来应对更广阔的开发需求。

目前有代表性的实时框架有:

- Meteor
- MEAN.io
- Derby
- SocketStream

不过说实话，目前能看到的实时通信的应用场景其实不多，其中大多集中于聊天室、to-do、实时图表、在线游戏等领域。其他领域使用实时特性不但没必要，而且是对服务器资源的浪费。因此目前是否要采用实时框架，要看具体的项目而定。

以上基本就是Node.js Web框架的现状了，相信看到这里，对于选择何种框架读者已经心里有数了吧。最后再介绍一个容易搞混的概念，和解释一下我的选择。

YEOMAN?

第一次见到这个词，我还以为它和MEAN有什么联系。事实上，它们是截然不同的两个东西。YEOMAN由YO（脚手架）、grunt（构建工具）、bower（包管理器），它代表的是一种工作流，与框架开发的思维方式完全不同。具体的介绍可见这里。

YEOMAN能够和框架达到类似的目的，都是为构建一个Web应用做好准备，但是要不要采用YEOMAN，则是见仁见智。我个人的看法是，学习

YEOMAN本身就需要不少时间，并且有一定的学习门槛。至少在目前，使用框架开发还是相对经济的，而如果以后YEOMAN这种模式推广开来，再来学习也 不迟，更何况有一定的Node.js项目经验之后再来学习YEOMAN要轻松很多。

事实上，我还是很认可YEOMAN这种Generator+package Manager的模式的，这是因为Node.js本身崇尚微模块的 概念，即无论是多么小的功能，都将它们模块化，甚至大的模块也要拆分成小的模块，然后通过搭积木的方式来构建应用。这样能够彻底的解耦，对于不容易调试的 Javascript来说，也有助于定位和修复应用中的问题。Generator就是这种理念催生下的产物，通过选择不同的配置和选项，将积木搭起来。不 过对于这种模式目前大家也还处于实验当中，不急于进行实际应用。

为什么我选择了Hackathon Starter?

在我的个人项目中，使用的是Hackathon Starter，一个Node.js Web应用脚手架。

我使用它的原因是，要求高度可配置，同时又讨厌写一些配置的代码，因此它对于我来说是很好的选择。一些全栈框架对我来说，封装过多，将原生的 Node.js/Express API隐藏掉了，要使用还需要一定的学习成本。而Express这样的框架又太过简洁，在实际的项目中使用还需要大量的插件和配置，而这些在 Hackathon Starter中都已经帮我们做好了，同时还有一些示例代码以供学习，对于新人来说非常友好，可以避免过多的挫折感。

上面一段可以看做是免费为Hackathon Starter做的广告吧，开源项目需要宣传和布道才能让更多人所关注。

最后，本文里的框架大多来源于nodeframework网站，本文可以看做是该站的注释版，在扫清我自己的一些疑惑的同时，也希望对读者有所帮助。

原文链接：http://idlelife.org/archives/516

JavaScript 运行机制详解：再谈 Event Loop

作者：阮一峰

一、为什么JavaScript是单线程?

JavaScript语言的一大特点就是单线程，也就是说，同一个时间只能做一件事。那么，为什么JavaScript不能有多个线程呢？这样能提高效率啊。

JavaScript的单线程，与它的用途有关。作为浏览器脚本语言，JavaScript的主要用途是与用户互动，以及操作DOM。这决定了它只 能是单线程，否则会带来很复杂的同步问题。比如，假定JavaScript同时有两个线程，一个线程在某个DOM节点上添加内容，另一个线程删除了这个节点，这时浏览器应该以哪个线程为准?

所以，为了避免复杂性，从一诞生，JavaScript就是单线程，这已经成了这门语言的核心特征，将来也不会改变。

为了利用多核CPU的计算能力，HTML5提出Web Worker标准，允许JavaScript脚本创建多个线程，但是子线程完全受主线程控制，且不得操作DOM。所以，这个新标准并没有改变JavaScript单线程的本质。

二、任务队列

单线程就意味着，所有任务需要排队，前一个任务结束，才会执行后一个任务。如果前一个任务耗时很长，后一个任务就不得不一直等着。

如果排队是因为计算量大，CPU忙不过来，倒也算了，但是很多时候CPU是闲着的，因为IO设备（输入输出设备）很慢（比如Ajax操作从网络读取数据），不得不等着结果出来，再往下执行。

JavaScript语言的设计者意识到，这时主线程完全可以不管IO设备，挂起处于等待中的任务，先运行排在后面的任务。等到IO设备返回了结果，再回过头，把挂起的任务继续执行下去。

于是，所有任务可以分成两种，一种是同步任务（synchronous），另一种是异步任务（asynchronous）。同步任务指的是，在主线 程上排队执行的任务，只有前一个任务执行完毕，才能执行后一个任务；异步任务指的是，不进入主线程、而进入"任务队列"（task queue）的任务，只有"任务队列"通知主线程，某个异步任务可以执行了，该任务才会进入主线程执行。

具体来说，异步执行的运行机制如下。（同步执行也是如此，因为它可以被视为没有异步任务的异步执行。）

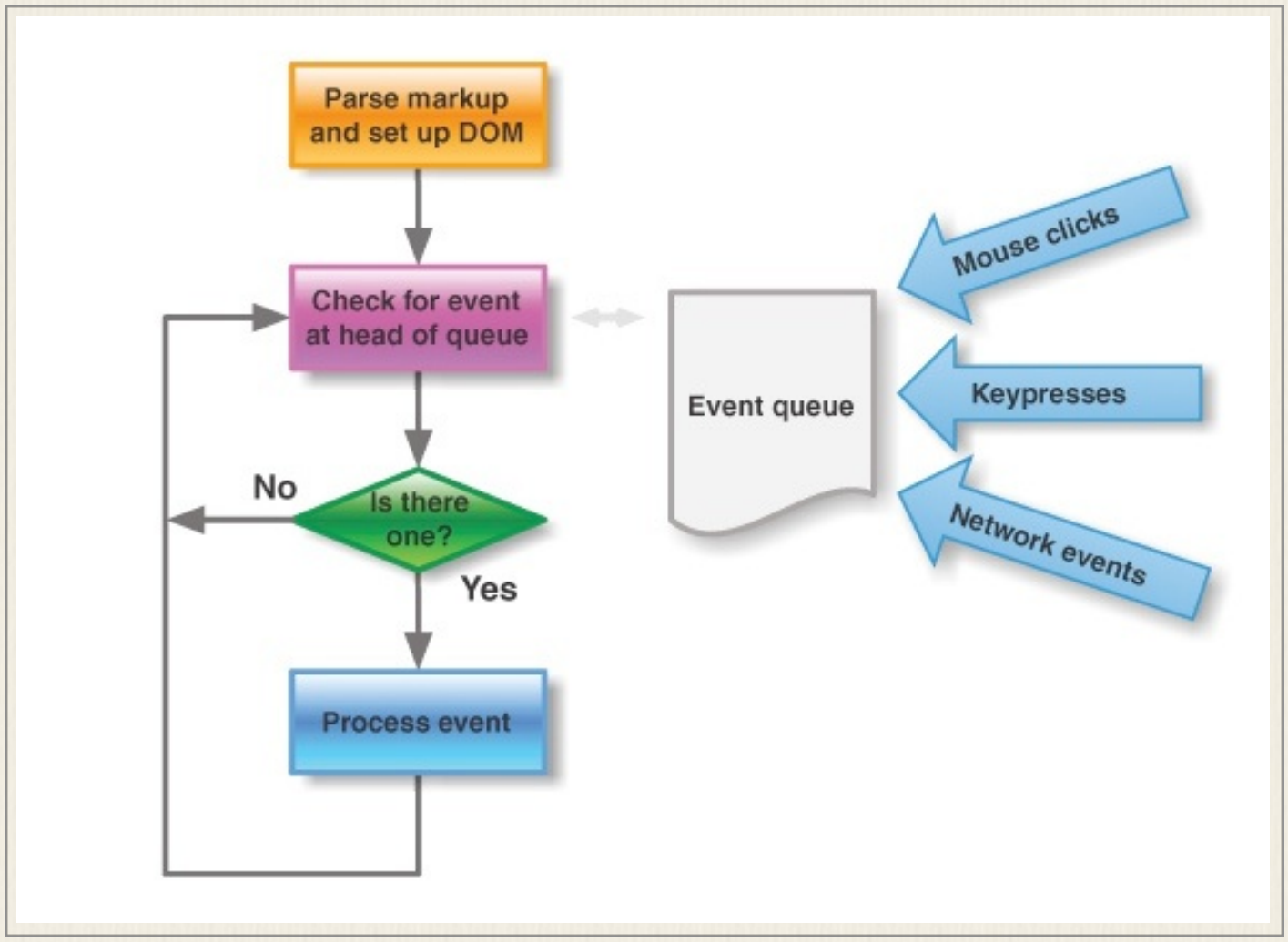
（1）所有同步任务都在主线程上执行，形成一个执行栈（execution context stack）。

（2）主线程之外，还存在一个"任务队列"（task queue）。只要异步任务有了运行结果，就在"任务队列"之中放置一个事件。

（3）一旦"执行栈"中的所有同步任务执行完毕，系统就会读取"任务队列"，看看里面有哪些事件。那些对应的异步任务，于是结束等待状态，进入执行栈，开始执行。

（4）主线程不断重复上面的第三步。

下图就是主线程和任务队列的示意图。



只要主线程空了，就会去读取"任务队列"，这就是JavaScript的运行机制。这个过程会不断重复。

三、事件和回调函数

"任务队列"是一个事件的队列（也可以理解成消息的队列），IO设备完成一项任务，就在"任务队列"中添加一个事件，表示相关的异步任务可以进入"执行栈"了。主线程读取"任务队列"，就是读取里面有哪些事件。

"任务队列"中的事件，除了IO设备的事件以外，还包括一些用户产生的事件（比如鼠标点击、页面滚动等等）。只要指定过回调函数，这些事件发生时就会进入"任务队列"，等待主线程读取。

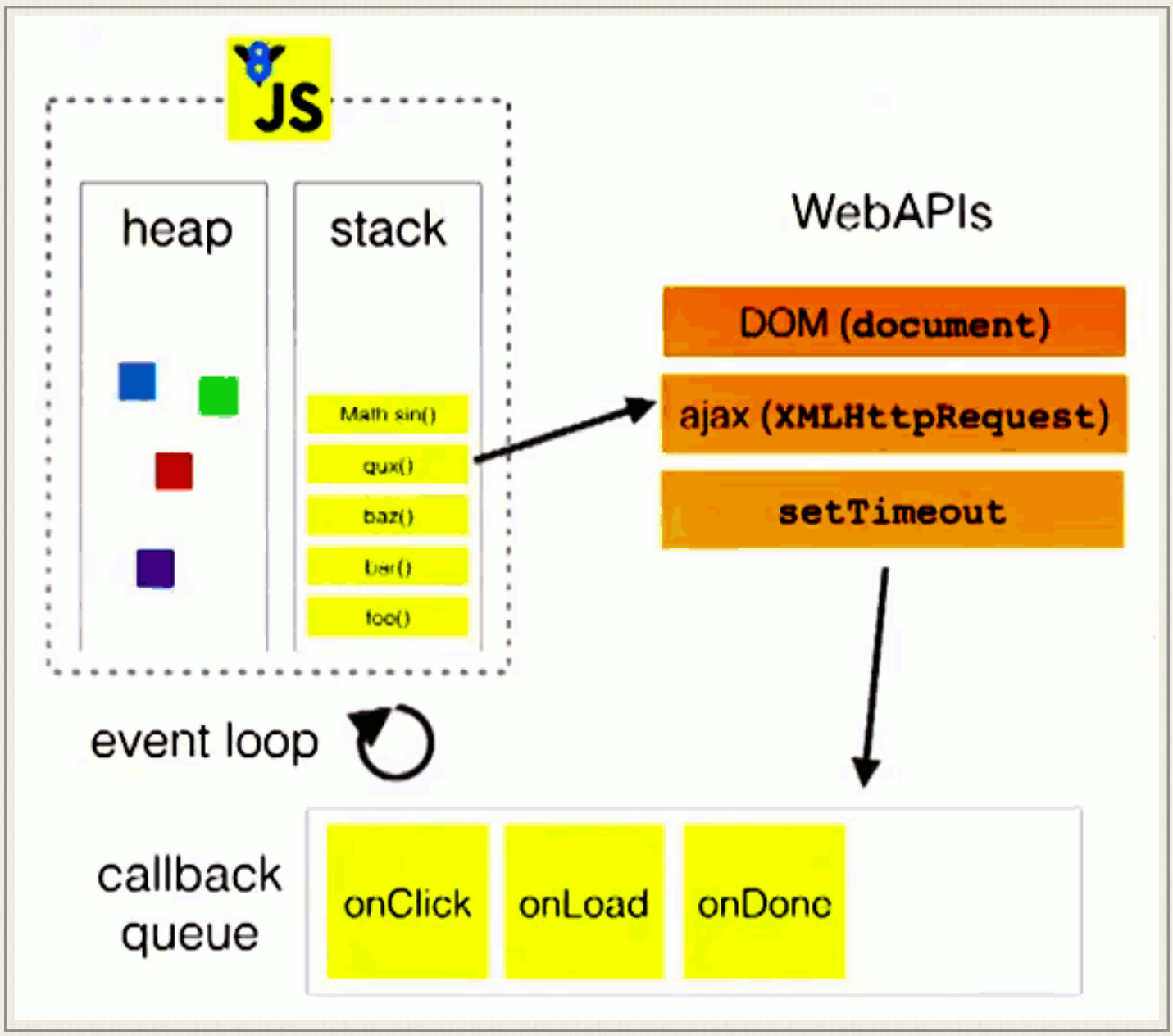
所谓"回调函数"（callback），就是那些会被主线程挂起来的代码。异步任务必须指定回调函数，当主线程开始执行异步任务，就是执行对应的回调函数。

"任务队列"是一个先进先出的数据结构，排在前面的事件，优先被主线程读取。主线程的读取过程基本上是自动的，只要执行栈一清空，"任务队列"上第 一位的事件就自动进入主线程。但是，由于存在后文提到的"定时器"功能，主线程首先要检查一下执行时间，某些事件只有到了规定的时间，才能返回主线程。

四、Event Loop

主线程从"任务队列"中读取事件，这个过程是循环不断的，所以整个的这种运行机制又称为Event Loop（事件循环）。

为了更好地理解Event Loop，请看下图（转引自Philip Roberts的演讲《Help, I'm stuck in an event-loop》）。



上图中，主线程运行的时候，产生堆（heap）和栈（stack），栈中的代码调用各种外部API，它们在"任务队列"中加入各种事件（click，load，done）。只要栈中的代码执行完毕，主线程就会去读取"任务队列"，依次执行那些事件所对应的回调函数。

执行栈中的代码（同步任务），总是在读取"任务队列"（异步任务）之前执行。请看下面这个例子。

```
var req = new XMLHttpRequest();

req.open('GET', url);

req.onload = function (){};

req.onerror = function (){};
```

```
req.send();
```

上面代码中的req.send方法是Ajax操作向服务器发送数据，它是一个异步任务，意味着只有当前脚本的所有代码执行完，系统才会去读取"任务队列"。所以，它与下面的写法等价。

```
var req = new XMLHttpRequest();
req.open('GET', url);
req.send();
req.onload = function (){};
req.onerror = function (){};
```

也就是说，指定回调函数的部分（onload和onerror），在send()方法的前面或后面无关紧要，因为它们属于执行栈的一部分，系统总是执行完它们，才会去读取"任务队列"。

五、定时器

除了放置异步任务的事件，"任务队列"还可以放置定时事件，即指定某些代码在多少时间之后执行。这叫做"定时器"（timer）功能，也就是定时执行的代码。

定时器功能主要由setTimeout()和setInterval()这两个函数来完成，它们的内部运行机制完全一样，区别在于前者指定的代码是一次性执行，后者则为反复执行。以下主要讨论setTimeout()。

setTime-
out()接受两个参数，第一个是回调函数，第二个是推迟执行的毫秒数。

```
console.log(1);
setTimeout(function(){console.log(2);},1000);
console.log(3);
```

上面代码的执行结果是1，3，2，因为setTimeout()将第二行推迟到1000毫秒之后执行。

如果将setTimeout()的第二个参数设为0，就表示当前代码执行完（执行栈清空）以后，立即执行（0毫秒间隔）指定的回调函数。

```
setTimeout(function(){console.log(1);}, 0);

console.log(2);
```

上面代码的执行结果总是2，1，因为只有在执行完第二行以后，系统才会去执行"任务队列"中的回调函数。

总之，setTimeout(fn,0)的含义是，指定某个任务在主线程最早可得的空闲时间执行，也就是说，尽可能早得执行。它在"任务队列"的尾部添加一个事件，因此要等到同步任务和"任务队列"现有的事件都处理完，才会得到执行。

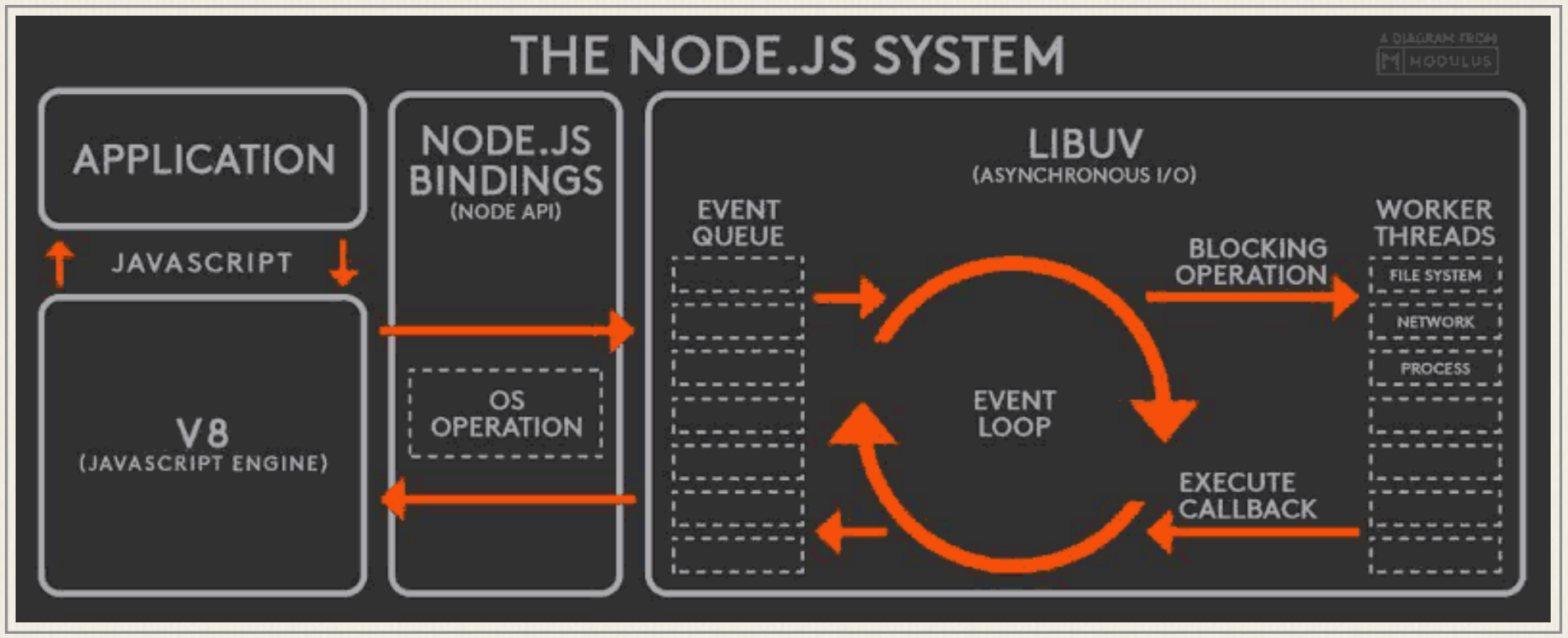
HTML5标准规定了setTimeout()的第二个参数的最小值（最短间隔），不得低于4毫秒，如果低于这个值，就会自动增加。在此之前，老版 本的浏览器都将最短间隔设为10毫秒。另外，对于那些DOM的变动（尤其是涉及页面重新渲染的部分），通常不会立即执行，而是每16毫秒执行一次。这时使 用requestAnimationFrame()的效果要好于setTimeout()。

需要注意的是，setTimeout()只是将事件插入了"任务队列"，必须等到当前代码（执行栈）执行完，主线程才会去执行它指定的回调函数。要是当前代码耗时很长，有可能要等很久，所以并没有办法保证，回调函数一定会在setTimeout()指定的时间执行。

六、Node.js的Event Loop

Node.js也是单线程的Event Loop，但是它的运行机制不同于浏览器环境。

请看下面的示意图（作者@BusyRich）。



根据上图，Node.js的运行机制如下。

（1）V8引擎解析JavaScript脚本。

（2）解析后的代码，调用Node API。

（3）libuv库负责Node API的执行。它将不同的任务分配给不同的线程，形成一个Event Loop（事件循环），以异步的方式将任务的执行结果返回给V8引擎。

（4）V8引擎再将结果返回给用户。

除了setTimeout和setInterval这两个方法，Node.js还提供了另外两个与"任务队列"有关的方法：process.nextTick和setImmediate。它们可以帮助我们加深对"任务队列"的理解。

process.nextTick方法可以在当前"执行栈"的尾部----下一次Event Loop（主线程读取"任务队列"）之前----触发回调函数。也就是说，它指定的任务总是发生在所有异步任务之前。setImmediate方法则是在 当前"任务队列"的尾部添加事件，也就是说，它指定的任务总是在下一次Event Loop时执行，这与setTimeout(fn, 0)很像。请看下面的例子（via StackOver-flow）。

```
process.nextTick(function A() {
  console.log(1);
```

```
  process.nextTick(function B(){console.log(2);});
});

setTimeout(function timeout() {
  console.log('TIMEOUT FIRED');
}, 0)
// 1
// 2
// TIMEOUT FIRED
```

上面代码中，由于process.nextTick方法指定的回调函数，总是在当前"执行栈"的尾部触发，所以不仅函数A比setTimeout指 定的回调函数timeout先执行，而且函数B也比timeout先执行。这说明，如果有多个process.nextTick语句（不管它们是否嵌 套），将全部在当前"执行栈"执行。

现在，再看setImmediate。

```
setImmediate(function A() {
  console.log(1);
  setImmediate(function B(){console.log(2);});
});

setTimeout(function timeout() {
  console.log('TIMEOUT FIRED');
}, 0);
```

上面代码中，setImmediate与setTimeout(fn,0)各自添加了一个回调函数A和timeout，都是在下一次Event Loop触发。那么，哪个回调函数先执行

呢？答案是不确定。运行结果可能是1--TIMEOUT FIRED--2，也可能是TIMEOUT FIRED--1--2。

令人困惑的是，Node.js文档中称，setImmediate指定的回调函数，总是排在setTimeout前面。实际上，这种情况只发生在递归调用的时候。

```
setImmediate(function (){
  setImmediate(function A() {
    console.log(1);
    setImmediate(function B(){console.log(2);});
  });

  setTimeout(function timeout() {
    console.log('TIMEOUT FIRED');
  }, 0);
});
// 1
// TIMEOUT FIRED
// 2
```

上面代码中，setImmediate和setTimeout被封装在一个setImmediate里面，它的运行结果总是1--TIMEOUT FIRED--2，这时函数A一定在timeout前面触发。至于2排在TIMEOUT FIRED的后面（即函数B在timeout后面触发），是因为setImmediate总是将事件注册到下一轮Event Loop，所以函数A和timeout是在同一轮Loop执行，而函数B在下一轮Loop执行。

我们由此得到了process.nextTick和setImmediate的一个重要区别：多个process.nextTick语句总是在当 前"执行栈"一次执行完，多个setImmediate可能则需要多次loop才能执行完。事实上，这正是Node.js 10.0版添加

setImmediate方法的原因，否则像下面这样的递归调用process.nextTick，将会没完没了，主线程根本不会去读取" 事件队列"！

```
process.nextTick(function foo() {
  process.nextTick(foo);
});
```

事实上，现在要是你写出递归的process.nextTick，Node.js会抛出一个警告，要求你改成setImmediate。

另外，由于process.nextTick指定的回调函数是在本次"事件循环"触发，而setImmediate指定的是在下次"事件循环"触发，所以很显然，前者总是比后者发生得早，而且执行效率也高（因为不用检查"任务队列"）。

原文链接：http://www.ruanyifeng.com/blog/2014/10/event-loop.html

抽象语法树在 JavaScript 中的应用

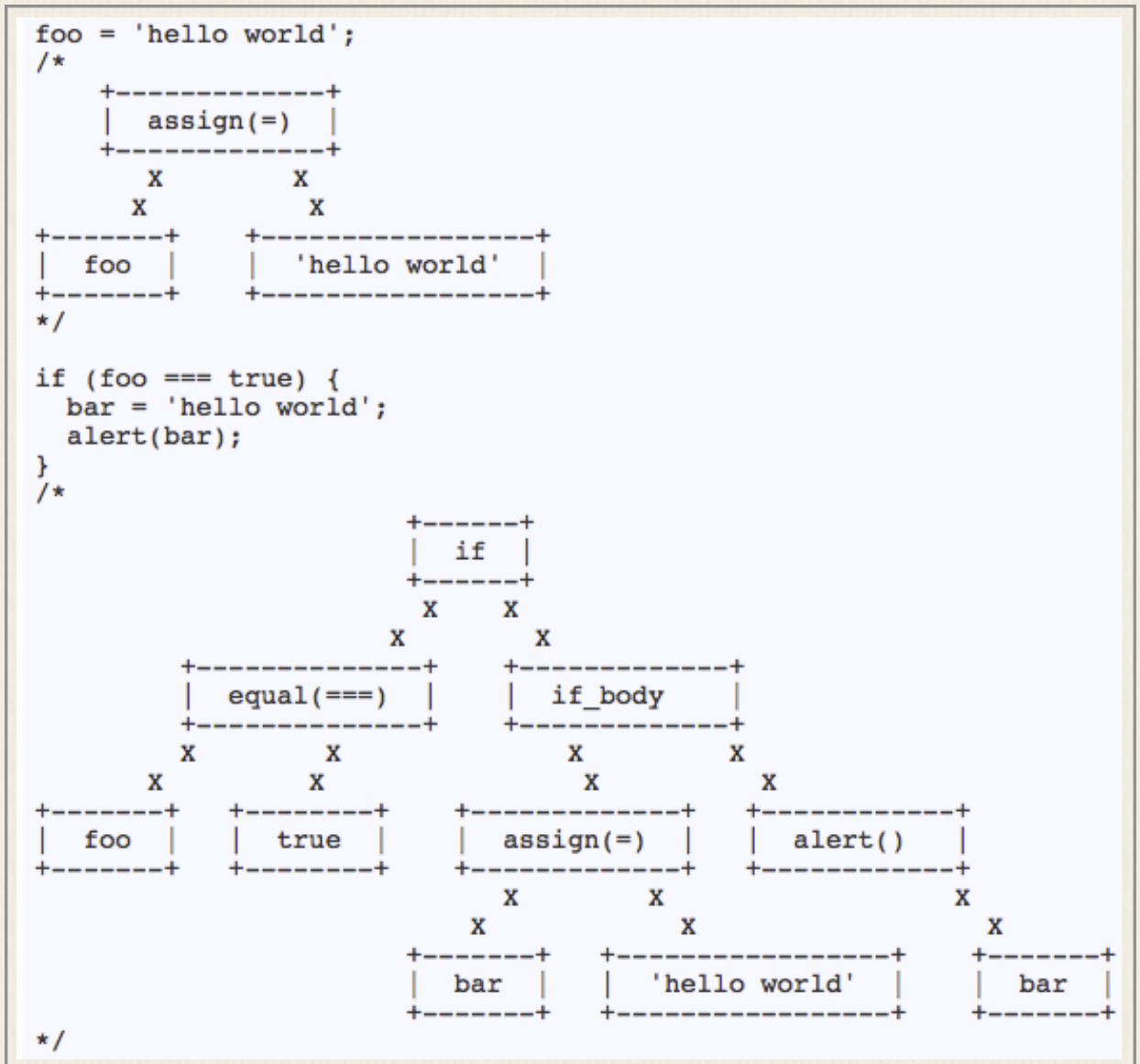
作者：美团技术团队

抽象语法树是什么

在计算机科学中，抽象语法树（abstract syntax tree 或者缩写为 AST），或者语法树（syntax tree），是源代码的抽象语法结构的树状表现形式，这里特指编程语言的源代码。树上的每个节点都表示源代码中的一种结构。之所以说语法是「抽象」的，是因为这里的语法并不会表示出真实语法中出现的每个细节。1

果然比较抽象，不如先看几个例子：

抽象语法树举例



从上述两个例子可以看出，抽象语法树是将源代码根据其语法结构，省略一些细节（比如：括号没有生成节点），抽象成树形表达。

抽象语法树在计算机科学中有很多应用，比如编译器、IDE、压缩优化代码等。下面介绍一下抽象语法树在 JavaScript 中的应用。

JavaScript 抽象语法树

构造 JavaScript 抽象语法树有多种工具，比如 v8、SpiderMonkey、UglifyJS 等，这里重点介绍 UglifyJS。

UglifyJS

UglifyJS 是使用最广的 JavaScript 压缩工具之一，而且自身也是用 JavaScript 写的，使用它的方法很简单（需要 nodejs 环境）：

首先全局安装：

[sudo]npm install -g uglify-js

然后就可以使用了：

uglifyjs -m srcFileName.js -o destFileName.min.js

关于 UglifyJS 的用法这里就不多介绍了，我们要做的是一些更有趣的事情。

UglifyJS Tools

UglifyJS 提供了一些工具用于分析 JavaScript 代码，包括：

- parser，把 JavaScript 代码解析成抽象语法树
- code generator，通过抽象语法树生成代码
- mangler，混淆 JavaScript 代码
- scope analyzer，分析变量定义的工具
- tree walker，遍历树节点
- tree transformer，改变树节点

生成抽象语法树

使用 UglifyJS 生成抽象语法树很简单：

首先安装 UglifyJS 为 npm 包：

npm install uglify-js --save-dev

然后使用 parse 方法即可：

```
var UglifyJS = require('uglify-js');
var ast = UglifyJS.parse('function sum(foo, bar){ return foo + bar; }');
```

这样生成的 ast 即为那一段代码的抽象语法树。那么我们怎么使用呢?

使用 mangler 压缩代码

使用 mangler 可以通过将局部变量都缩短成一个字符来压缩代码。

```
var UglifyJS = require('uglify-js');
var ast = UglifyJS.parse('function sum(foo, bar){ return foo + bar; }');
ast.figure_out_scope();
ast.mangle_names();
console.log(ast.print_to_string());
// function sum(a,b){return a+b}
```

使用 walker 遍历抽象语法树

使用 walker 可以遍历抽象语法树，这种遍历是深度遍历。

```
var UglifyJS = require('uglify-js');
var ast = UglifyJS.parse('function sum(foo, bar){ return foo + bar; }');
ast.figure_out_scope();
ast.walk(new UglifyJS.TreeWalker(function(node) {
    console.log(node.print_to_string());
}));
```

```
/*
function sum(foo,bar){return foo+bar}
function sum(foo,bar){return foo+bar}
sum
foo
bar
return foo+bar
foo+bar
foo
bar
*/
```

UglifyJS 已经提供了直接压缩代码的脚本，walker 看上去貌似也没啥用，那么这些工具有什么使用场景呢?

抽象语法树的应用

利用抽象语法树重构 JavaScript 代码

假如我们有重构 JavaScript 的需求，它们就派上用场啦。

下面考虑这样一个需求:

我们知道，parseInt 用于将字符串变成整数，但是它有第二个参数，表示以几进制识别字符串，若没有传第二个参数，则会自行判断，比如:

```
parseInt('10.23');     // 10            转换成正整数
parseInt('10abc');     // 10            忽略其他字符
parseInt('10', 10);    // 10            转换成十进制
parseInt('10', 2);     // 2             转换成二进制
parseInt('0123');      // 83 or 123     不同浏览器不一样，低版本浏览器会转换成八进制
```

```
parseInt('0x11');     // 17          转换成十六进制
```

因为有一些情况是和我们预期不同的，所以建议任何时候都加上第二个参数。

下面希望有一个脚本，查看所有 parseInt 有没有第二个参数，没有的话加上第二个参数 10，表示以十进制识别字符串。

使用 UglifyJS 可以实现此功能：

```
#! /usr/bin/env node


var U2 = require("uglify-js");


function replace_parseint(code) {
    var ast = U2.parse(code);
    // accumulate `parseInt()` nodes in this array
    var parseint_nodes = [];
    ast.walk(new U2.TreeWalker(function(node){
        if (node instanceof U2.AST_Call
            && node.expression.print_to_string() === 'parseInt'
            && node.args.length === 1) {
            parseint_nodes.push(node);
        }
    }));
    // now go through the nodes backwards and replace code
    for (var i = parseint_nodes.length; --i >= 0;) {
        var node = parseint_nodes[i];
        var start_pos = node.start.pos;
```

```
        var end_pos = node.end.endpos;
        node.args.push(new U2.AST_Number({
            value: 10
        }));
        var replacement = node.print_to_string({ beautify: true });
        code = splice_string(code, start_pos, end_pos, replacement);
    }
    return code;
}

function splice_string(str, begin, end, replacement) {
    return str.substr(0, begin) + replacement + str.substr(end);
}

// test it

function test() {
    if (foo) {
      parseInt('12342');
    }
    parseInt('0012', 3);
}

console.log(replace_parseint(test.toString()));
```

```
/*
function test() {
    if (foo) {
      parseInt("12342", 10);
    }
    parseInt('0012', 3);
}
*/
```

在这里，使用了 walker 找到 parseInt 调用的地方，然后检查是否有第二个参数，没有的话，记录下来，之后根据每个记录，用新的包含第二个参数的内容替换掉原内容，完成代码的重构。

也许有人会问，这种简单的情况，用正则匹配也可以方便的替换，干嘛要用抽象语法树呢?

答案就是，抽象语法树是通过分析语法实现的，有一些正则无法（或者很难）做到的优势，比如，parseInt() 整个是一个字符串，或者在注释中，此种情况会被正则误判:

```
var foo = 'parseInt("12345")';
// parseInt("12345");
```

抽象语法树在美团中的应用

在美团前端团队，我们使用 YUI 作为前端底层框架，之前面临的一个实际问题是，模块之间的依赖关系容易出现疏漏。比如:

```
YUI.add('mod1', function(Y) {
    Y.one('#button1').simulate('click');
    Y.Array.each(array, fn);
    Y.mod1 = function() {/**/};
```

```
}, '', {
    requires: [
        'node',
        'array-extras'
    ]
});
YUI.add('mod2', function(Y) {
    Y.mod1();
    // Y.io(uri, config);
}, '', {
    requires: [
        'mod1',
        'io'
    ]
});
```

以上代码定义了两个模块，其中 mod1 模拟点击了一下 id 为 button1 的元素，执行了 Y.Array.each，然后定义了方法 Y.mod1，最后声明了依赖 node 和 array-extras；mod2 执行了 mod1 中定义的方法，而 Y.io 被注释了，最后声明了依赖 mod1 和 io。

此处 mod1 出现了两个常见错误，一个是 simulate 是 Y.Node.prototype 上的方法，容易忘掉声明依赖 node-event-simulate3，另一个是 Y.Array 上只有部分方法需要依赖 array-extras，故此处多声明了依赖 array-extras4；mod2 中添加注释后，容易忘记删除原来写的依赖 io。

故正确的依赖关系应该如下：

```
YUI.add('mod1', function(Y) {
    Y.one('#button1').simulate('click');
```

```
    Y.Array.each(array, fn);
    Y.mod1 = function() {/**/};
}, '', {
    requires: [
        'node',
        'node-event-simulate'
    ]
});
YUI.add('mod2', function(Y) {
    Y.mod1();
    // Y.io(uri, config);
}, '', {
    requires: [
        'mod1'
    ]
});
```

为了使模块依赖关系的检测自动化，我们创建了模块依赖关系检测工具，它利用抽象语法树，分析出定义了哪些接口，使用了哪些接口，然后查找这些接口应该依赖哪些模块，进而找到模块依赖关系的错误，大致的过程如下：

1. 找到代码中模块定义（YUI.add）的部分
2. 分析每个模块内函数定义，变量定义，赋值语句等，找出符合要求（以 Y 开头）的输出接口（如 mod1 中的 Y.mod1）
3. 生成「接口 - 模块」对应关系

4. 分析每个模块内函数调用，变量使用等，找出符合要求的输入接口（如 mod2 中的 Y.one，Y.Array.each，Y.mod1）

5. 通过「接口 - 模块」对应关系，找到此模块应该依赖哪些其他模块

6. 分析 requires 中是否有错误

使用此工具，保证每次提交代码时，依赖关系都是正确无误的，它帮助我们实现了模块依赖关系检测的自动化。

总结

抽象语法树在计算机领域中应用广泛，以上仅讨论了抽象语法树在 JavaScript 中的一些应用，期待更多的用法等着大家去尝试和探索。

原文链接：http://tech.meituan.com/abstract-syntax-tree.html

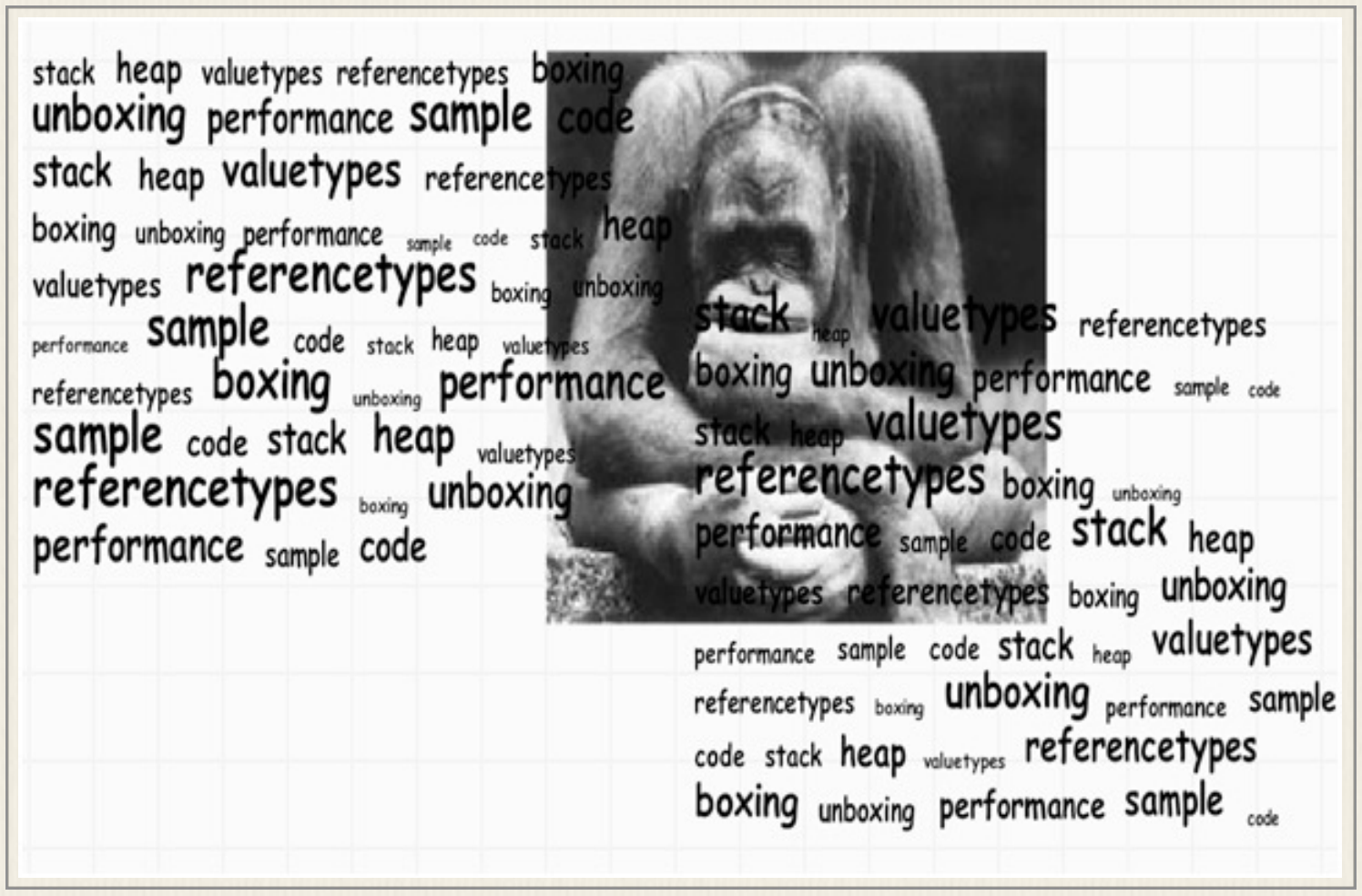
.NET中六个重要的概念：栈、堆、值类型、引用类型、装箱和拆箱

译者：周旭龙

一、概述

本文会阐述六个重要的概念：堆、栈、值类型、引用类型、装箱和拆箱。本文首先会通过阐述当你定义一个变量之后系统内部发生的改变开始讲解，然后将关注点转移到存储双雄：堆和栈。之后，我们会探讨一下值类型和引用类型，并对有关于这两种类型的重要基础内容做一个讲解。

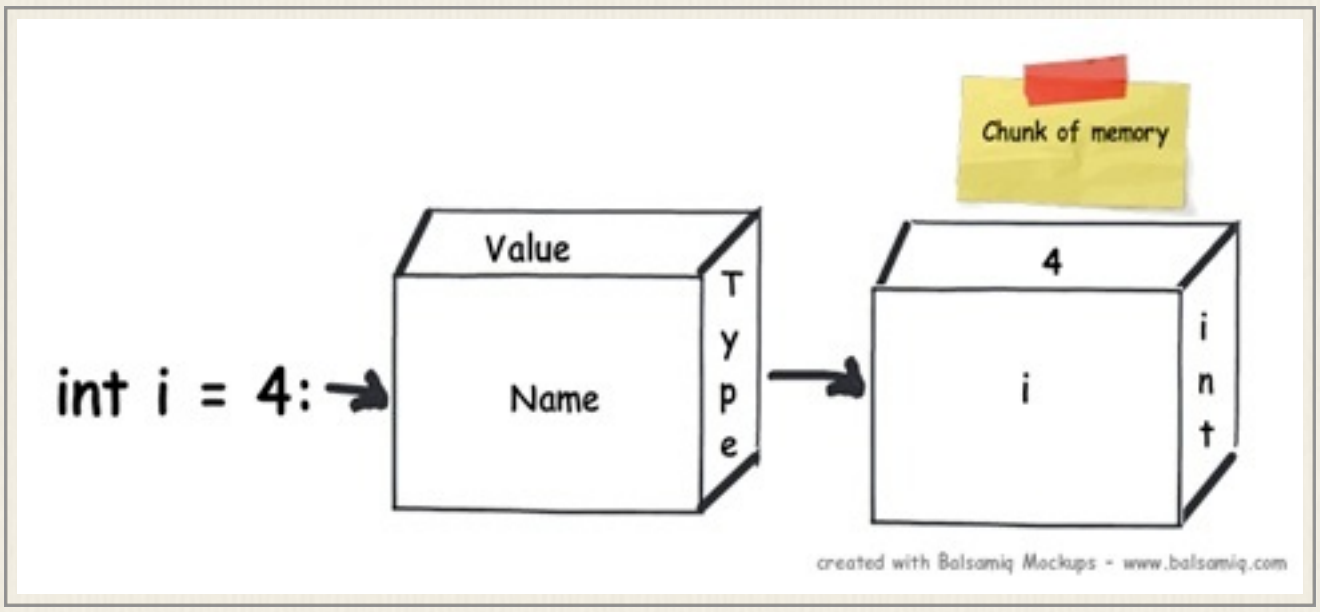
本文会通过一个简单的代码来展示在装箱和拆箱过程中所带来的性能上的影响，请各位仔细阅读。



二、当你声明一个变量背后发生了什么?

当你在一个.NET应用程序中定义一个变量时，在RAM中会为其分配一些内存块。这块内存有三样东西：变量的名称、变量的数据类型以及变量的值。

上面简单阐述了内存中发生的事情，但是你的变量究竟会被分配到哪种类型的内存取决于数据类型。在.NET中有两种可分配的内存：栈和堆。在接下来的几个部分中，我们会试着详细地来理解这两种类型的存储。



三、存储双雄：堆和栈

为了理解栈和堆，让我们通过以下的代码来了解背后到底发生了什么。

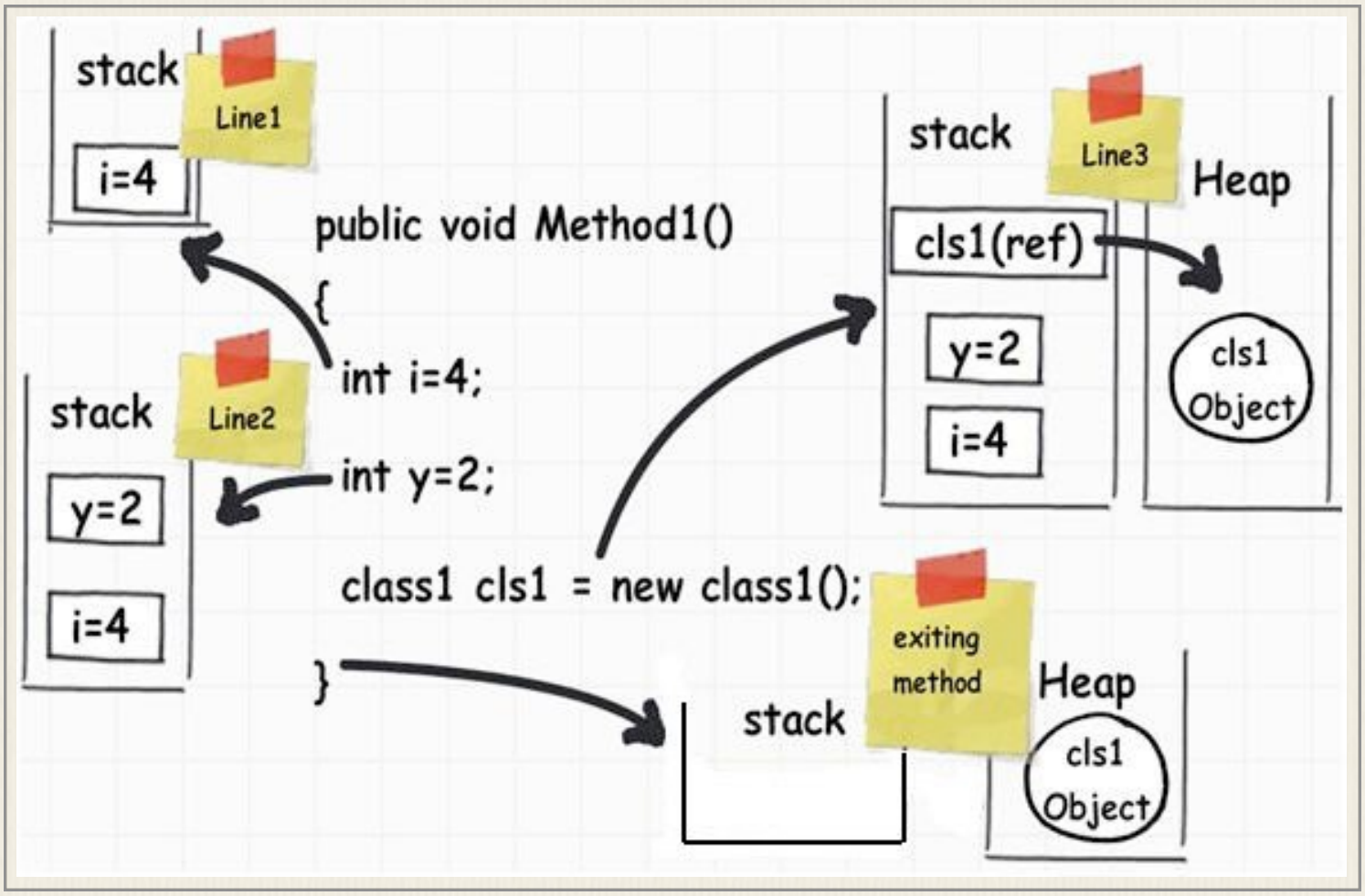
```
public void Method1()
{
    // Line 1
    int i=4;

    // Line 2
    int y=2;
```

```
    //Line 3
        class1 cls1 = new class1();
    }
```

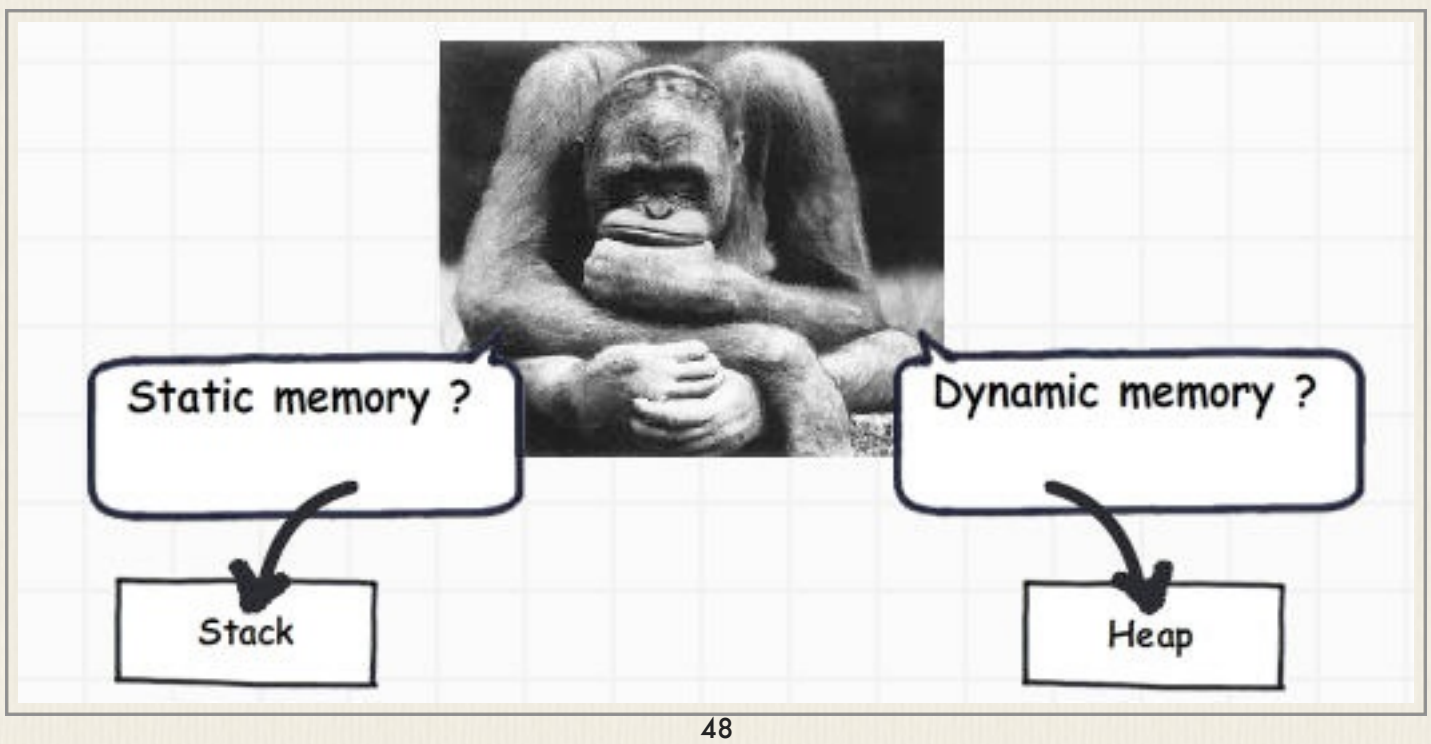
代码只有三行，现在我们可以一行一行地来了解到底内部是怎么来执行的。

- Line 1：当这一行被执行后，编译器会在栈上分配一小块内存。栈会在负责跟踪你的应用程序中是否有运行内存需要
- Line 2：现在将会执行第二步。正如栈的名字一样，它会将此处的一小块内存分配叠加在刚刚第一步的内存分 配的顶部。你可以认为栈就是一个一个叠加起来的房间或盒子。在栈中，数据的分配和解除都会通过LIFO (Last In First Out)即先进后出的逻辑规则进行。换句话说，也就是最先进入栈中的数据项有可能最后才会出栈。
- Line 3：在第三行中，我们创建了一个对象。当这一行被执行后，.NET会在栈中创建一个指针，而实际的对象将会存储到一个叫做“堆”的内存区域中。“堆”不会监测运行内存，它只是能够被随时访问到的一堆对象而已。不同于栈，堆用于动态内存的分配。
- 这里需要注意的另一个重要的点是对象的引用指针是分配在栈上的。 例如：声明语句 Class1 cls1; 其实并没有为Class1的实例分配内存，它只是在栈上为变量cls1创建了一个引用指针（并且将其默认职位null）。只有当其遇到new关键字时，它才会在堆上为对象分配内存。
- 离开这个Method1方法时（the fun）：现在执行控制语句开始离开方法体，这时所有在栈上为变量所分配的内存空间都会被清除。换句话说，在上面的示例中所有与int类型相关的变量将会按照“LIFO”后进先出的方式从栈中一个一个地出栈。
- 需要注意的是：这时它并不会释放堆中的内存块，堆中的内存块将会由垃圾回收器稍候进行清理。



现在我们许多的开发者朋友一定很好奇为什么会有两种不同类型的存储？我们为什么不能将所有的内存块分配只到一种类型的存储上？

如果你观察足够仔细，基元数据类型并不复杂，他们仅仅保存像 ‘int i = 0’这样的值。对象数据类型就复杂 了，他们引用其他对象或其他基元数据类型。换句话说，他们保存其他多个值的引用并且这些值必须一一地存储在内存中。对象类型需要的是动态内存而基元类型需 要静态内存。如果需求是动态内存的话，那么它将会在堆上为其分配内存，相反，则会在栈上为其分配。

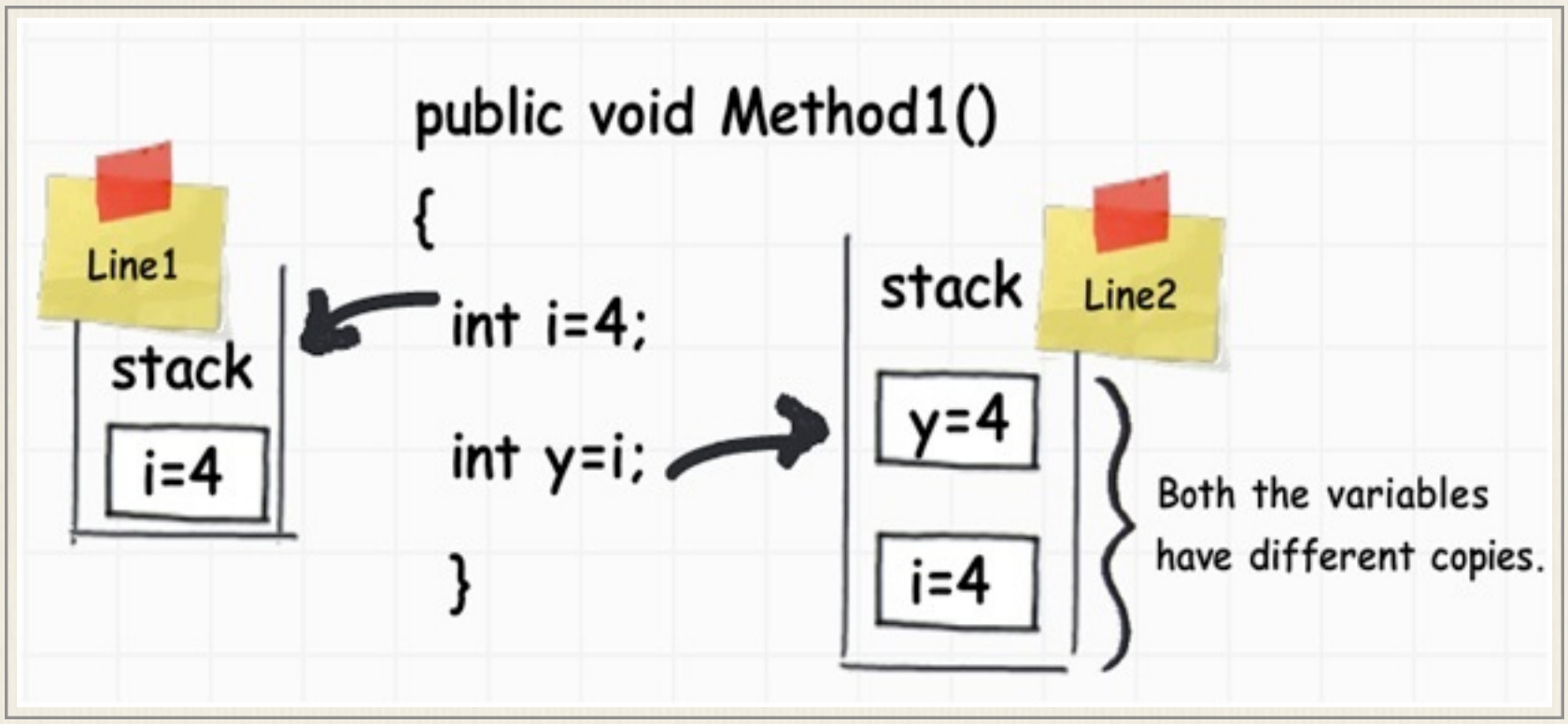


四、值类型和引用类型

既然我们已经了解了栈和堆的概念了，是时候了解值类型和引用类型的概念了。值类型将数据和内存都保存在同一位置，而一个引用类型则会有一个指向实际内存区域的指针。

通过下图，我们可以看到一个名为i的整形数据类型，它的值被赋值到另一个名为j的整形数据类型。他们的值都被存储到了栈上。

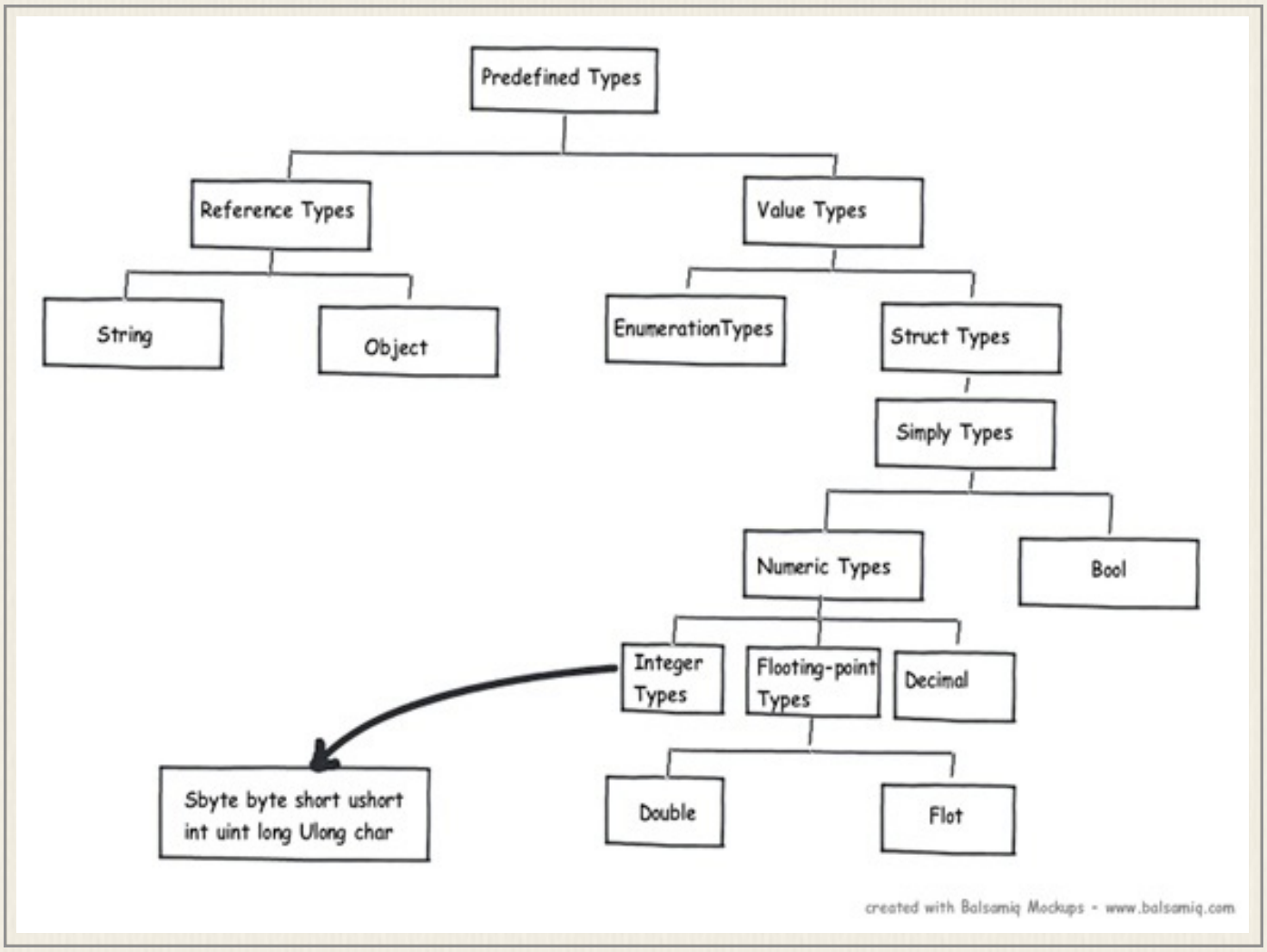
当我们将一个int类型的值赋值到另一个int类型的值时，它实际上是创建了一个完全不同的副本。换句话说，如果你改变了其中某一个的值，另一个不会发生改变。于是，这些种类的数据类型被称为“值类型”。



当我们创建一个对象并且将此对象赋值给另外一个对象时，他们彼此都指向了如下图代码段所示的内存中同一 块区域。因此，当我们将obj赋值给obj1时，他们都指向了堆中的同一块区域。换句话说，如果此时我们改变了其中任何一个，另一个都会受到影响，这也说 明了他们为何被称为“引用类型”。

五、哪些是值类型，哪些是引用类型?

在.NET中，变量是存储到栈还是堆中完全取决于其所属的数据类型。比如：‘String’或‘Object’属于引用类型，而其他.NET基元数据类型则会被分配到栈上。下图则详细地展示了在.NET预置类型中，哪些是值类型，哪些又是引用类型。



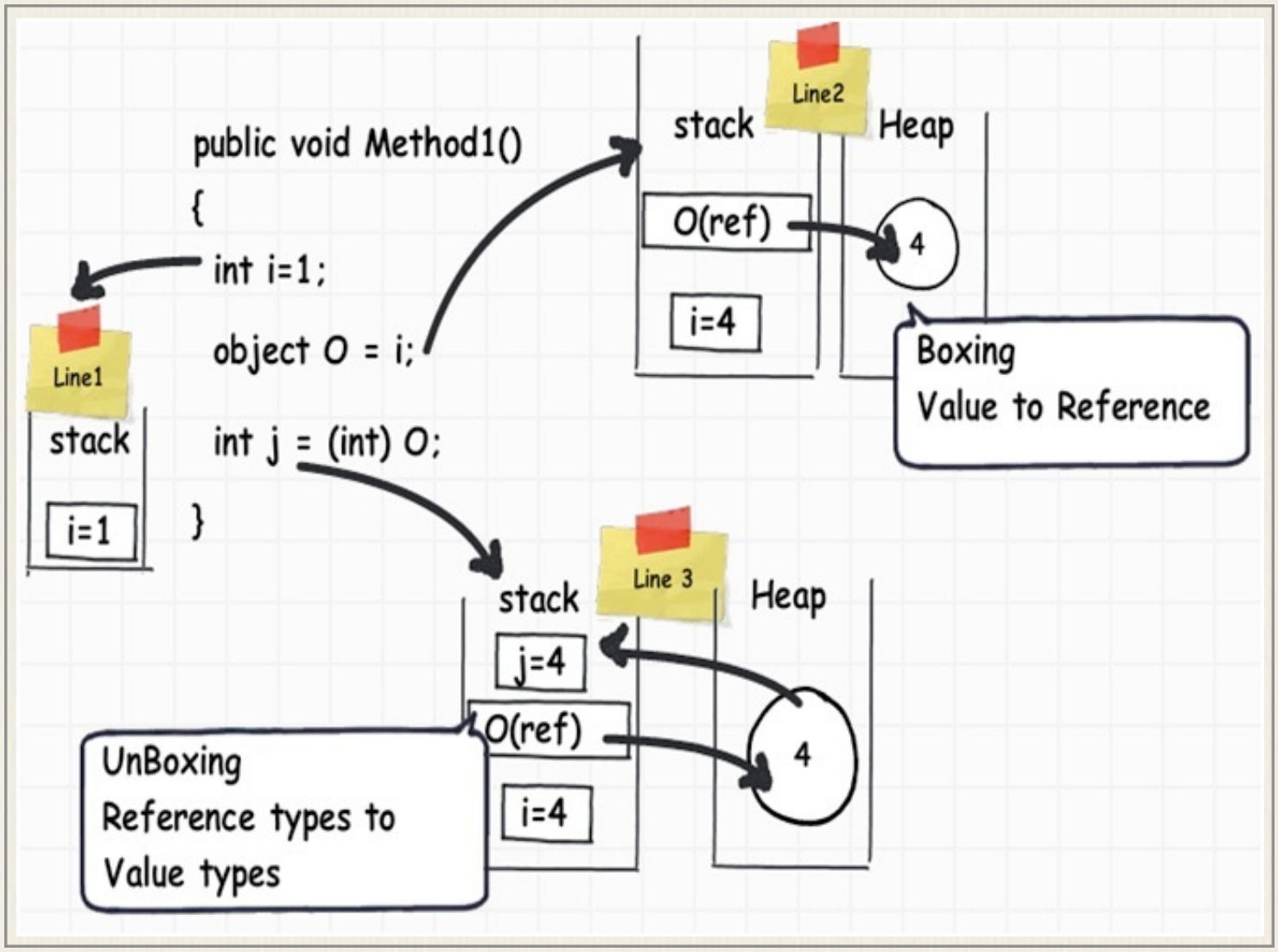
六、装箱和拆箱

现在，你已经有了不少的理论基础了。现在，是时候了解上面的知识在实际编程中的使用了。在应用中最大的一个意义就在于：理解数据从栈移动到堆的过程中所发生的性能消耗问题，反之亦然。

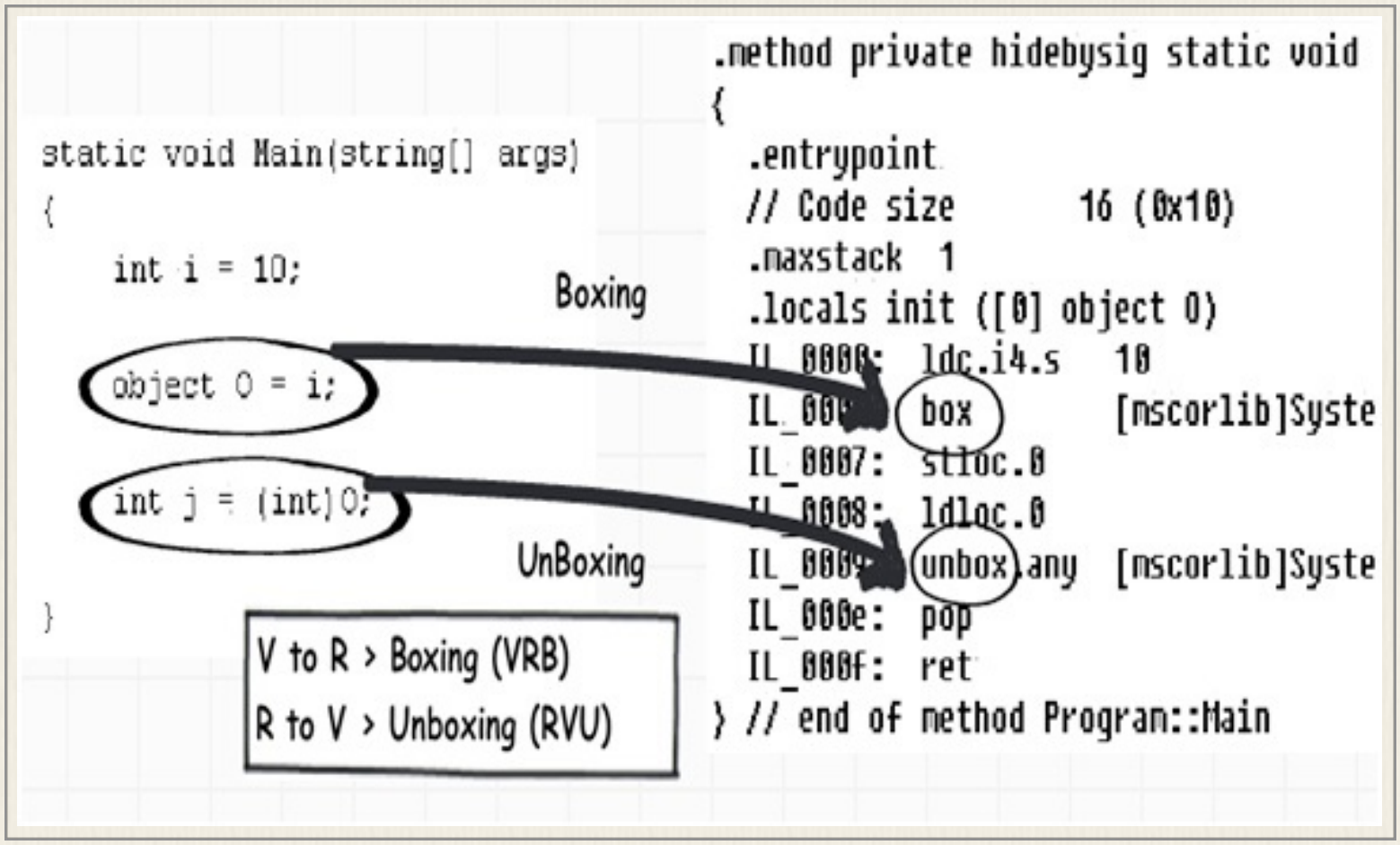
考虑一下以下的代码片段，当我们将一个值类型转换为引用类型，数据将会从栈移动到堆中。相反，当我们将一个引用类型转换为值类型时，数据也会从堆移动到栈中。

不管是在从栈移动到堆还是从堆中移动到栈上都会不可避免地对系统性能产生一些影响。

于是，两个新名词横空出世：当数据从值类型转换为引用类型的过程被称为“装箱”，而从引用类型转换为值类型的过程则被成为“拆箱”。



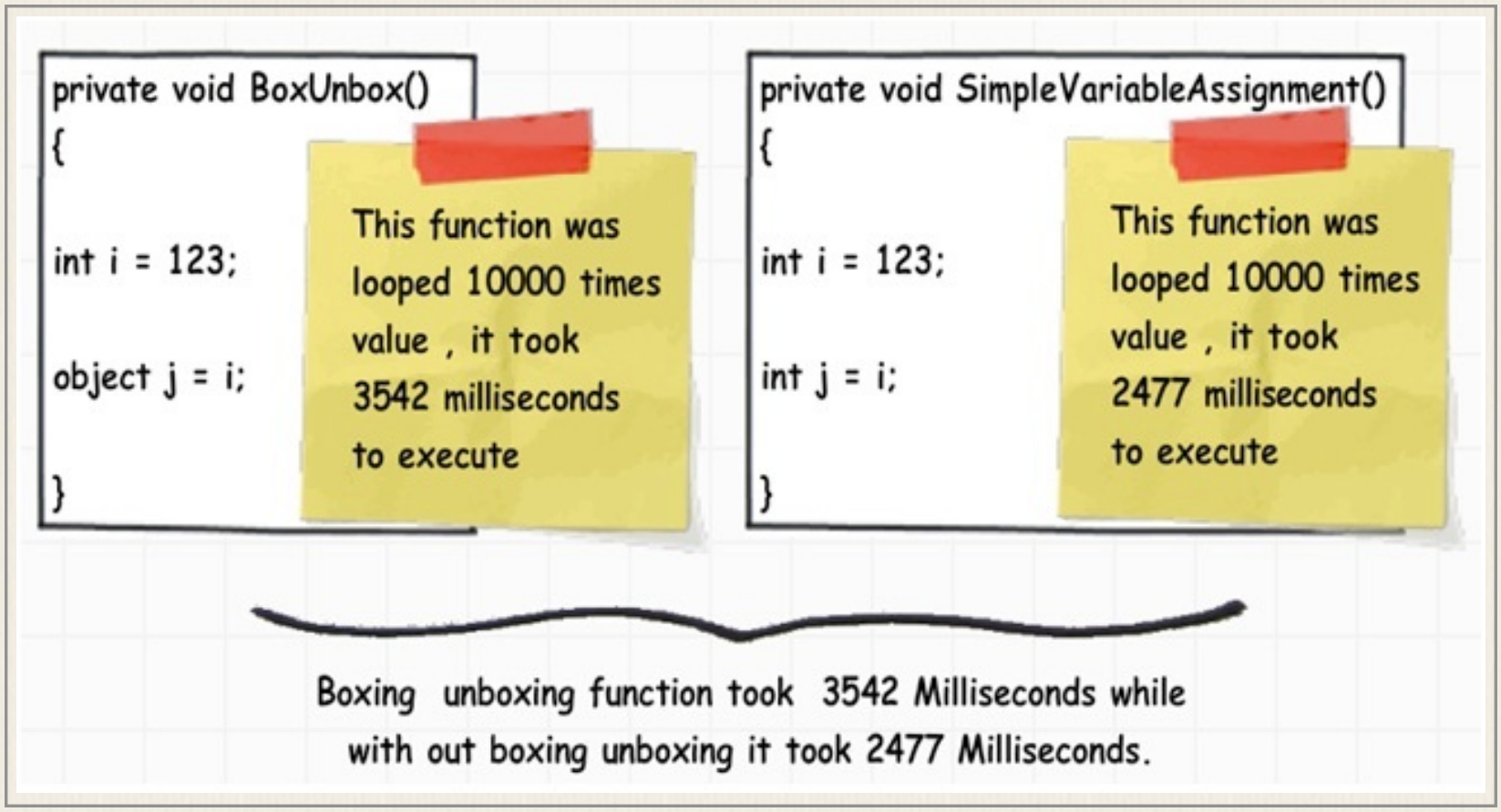
如果你编译一下上面这段代码并且在ILDASM（一个IL的反编译工具）中对其进行查看，你会发现在IL代码中，装箱和拆箱是什么样子的。下图则展示了示例代码被编译后所产生的IL代码。



七、装箱和拆箱的性能问题

为了弄明白到底装箱和拆箱会带来怎样的性能影响，我们分别循环运行10000次下图所示的两个函数方法。其中第一个方法中有装箱操作，另一个则没有。我们使用一个Stopwatch对象来监视时间的消耗。

具有装箱操作的方法花费了3542毫秒来执行完成，而没有装箱操作的方法只花费了2477毫秒，整 整相差了1秒多。而且，这个值也会因为循环次数的增加而增加。也就是说，我们要尽量避免装箱和拆箱操作。在一个项目中，如果你需要装箱和装箱，请仔细考虑 它是否是绝对必不可少的操作，如果不是，那么尽量不用。



虽然以上代码段没有展示拆箱操作，但其效果同样适用于拆箱。你可以通过写代码来实现拆箱，并且通过Stopwatch来测试其时间消耗。

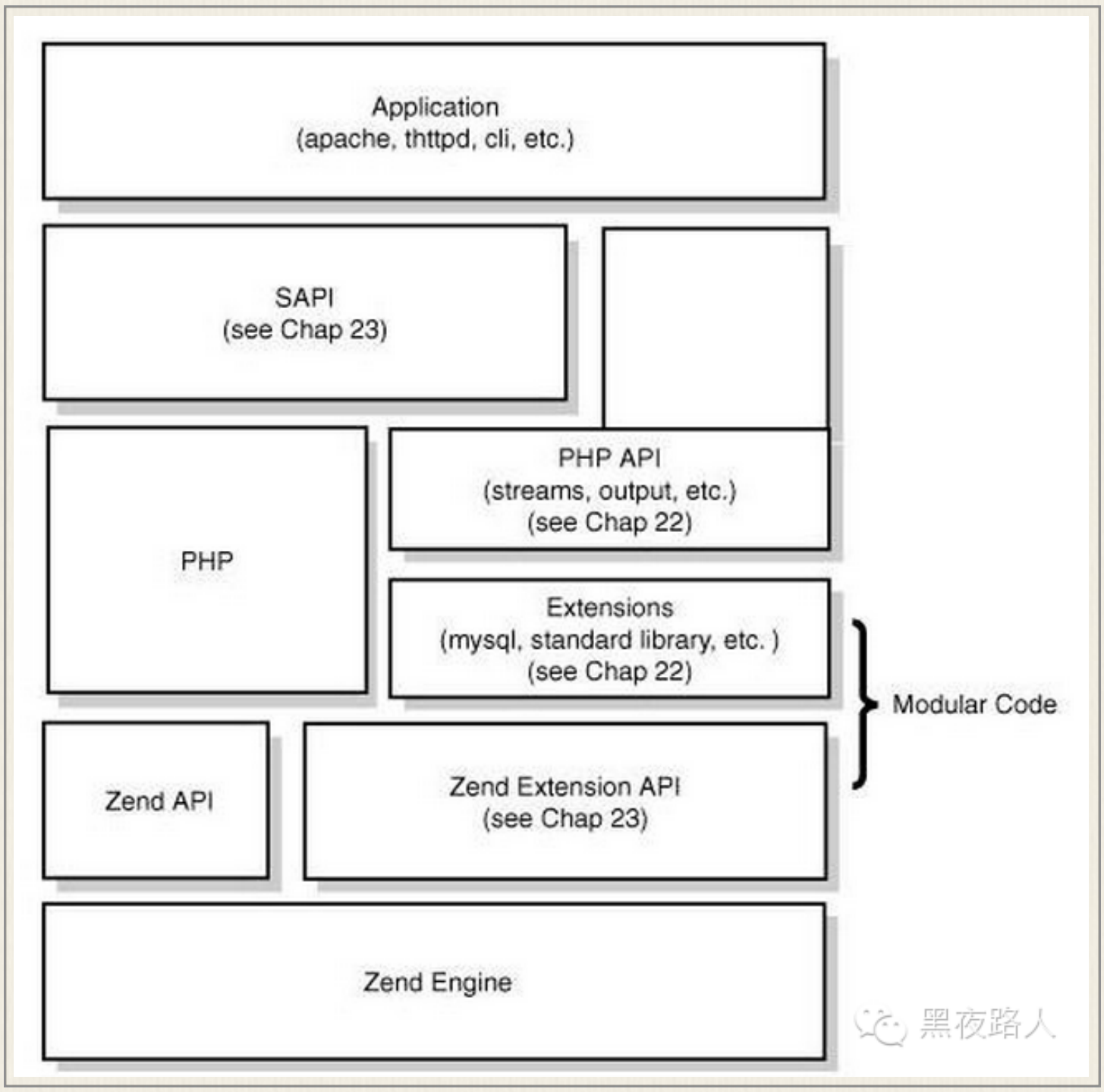
译文链接：http://www.cnblogs.com/edisonchou/p/3947170.html

原文链接：http://www.codeproject.com/Articles/76153/Six-important-NET-concepts-Stack-heap-value-types

PHP扩展开发入门

作者：吴峰（360电商技术组）

在我们编写自己的第一个php扩展之前，先了解一下php的整体架构和运行机制。



php的 架构如图1所示。其中一个重要的就是SAPI（服务器端应用编程端口），它使得PHP可以和其他应用进行数据交互，把外部错综复杂的外部环境进行抽象化， 为内部的php提供一套固定和统一的接口，使得php自身不受外部影响，保持一定的独立性。常见的SAPI有CGI，FastCGI，Shell的CLI，apache的mod_php5，IIS的ISAPI。

另外一个非常重要就是ZendEngine。Zend Engine是官方提供的PHP实现的核心，提供了语言实现上的基础设施，其他比较知名的还有facebook的hiphop实现。例如PHP的语法实 现，脚本的编译运行环境，扩展机制以及内存管理等。我们在后面编写php扩展时，也将基于Zend Engine。

PHP3时 代还是采用边解释边执行的运行方式，这种方式运行效率很受影响，其次代码整体耦合度比较高，可扩展性也不够好。因此随着php在web应用开发中的普及， 于是ZeevSuraski和Andi Gutmans决定重写代码以解决这两个问题，最终他们俩把该项技术的核心引擎命名为Zend Engine 。

Zend Engine最主要的特性就是把PHP的边解释边执行的运行方式改为先预编译(Compile)，再执行(Execute)。这两者的分开给 PHP 带来了革命性的变化：执行效率大幅提高。由于实行了功能分离，降低了模块间耦合度，可扩展性也大大增强。

目 前PHP的实现和Zend Engine之间的关系非常紧密，例如很多PHP扩展都是使用的Zend API，而Zend正是PHP语言本身的实现，PHP只是使用Zend这个内核来构建PHP语言的，而PHP扩展大都使用Zend API，这就导致PHP的很多扩展和Zend引擎耦合在一起了，后来才有PHP核心开发者就提出将这种耦合解开的建议。不过下面我们还下面在Zend Engine的基础上开始编写我们第一个简单的php扩展。

1.配置文件

每一个PHP扩展都至少需要一个配置文件和一个源文件。配置文件用来告诉编译器应该编译哪几个文件，以及编译本扩展是否需要的其它库文件。

在php源码文件夹的ext目录下创建一个新的文件，扩展的名字取作myfirst。然后在这个目录下创建一个config.m4文件，并输入以下内容：

```
PHP_ARG_ENABLE(
  myfirst,
  [Whether to enable the "myfirst" extension],
  [enable-myfirst    Enable"myfirst" extension support])
if test $PHP_Myfirst !="no"; then
  PHP_SUBST(Myfirst_SHARED_LIBADD)
  PHP_NEW_EXTENSION(myfirst, myfirst.c, $ext_shared)
fi
```

上 面PHP_ARG_ENABLE函数有三个参数，第一个参数是我们的扩展名(注意不用加引号)，第二个参数是当我们运行./configure脚本时显示 的内容，最后一个参数则是我们在调用./configure--help时显示的帮助信息。PHP_SUBST函数只是php官方对autoconf中 AC_SUBST函数的一层封装。PHP_NEW_EXTENSION函数声明了这个扩展的名称、需要的源文件名、扩展的编译形式。如果扩展使用了多个文 件，可以将文件名罗列在函数的参数里，如：PHP_NEW_EXTENSION(sample, sample.c sample2.c sample3.c, $ext_shared)最后的$ext_shared参数用来声明这个扩展为动态库，在php运行时动态加载的。

2.源文件

在完成了配置文件后，下面的就是完成扩展主逻辑的头文件和C文件。

头文件

```
//php_myfirst.h
#ifndef Myfirst_H
#define Myfirst_H
//加载config.h，如果配置了的话
#ifdef HAVE_CONFIG_H
#include "config.h"
#endif
//加载php头文件
#include "php.h"
#define phpext_myfirst_ptr &myfirst_module_entry
extern zend_module_entrymyfirst_module_entry;
#endif
```

C文件

```
//myfirst.c
#include "php_myfirst.h"
//module entry
zend_module_entrymyfirst_module_entry = {
#if ZEND_MODULE_API_NO >= 20010901
    STANDARD_MODULE_HEADER,
#endif
    "myfirst",//扩展名称
    NULL, /*Functions */
```

```
    NULL, /*MINIT */
    NULL, /*MSHUTDOWN */
    NULL, /*RINIT */
    NULL, /*RSHUTDOWN */
    NULL, /*MINFO */
#if ZEND_MODULE_API_NO >= 20010901
    "2.1",//扩展的版本
#endif
    STANDARD_MODULE_PROPERTIES
};

#ifdef COMPILE_DL_Myfirst
ZEND_GET_MODULE(myfirst)
#endif
```

3.扩展编译

准备好了扩展需要编译的源文件，接下来需要的便是把它们编译成目标文件了。

第一步：根据config.m4文件使用phpize生成一个configure脚本、Makefile等文件：

```
$ phpize
PHP Api Version: 20041225
Zend Module Api No: 20050617
```

Zend Extension Api No: 220050617

现在查看扩展所在的目录，会发现phpize程序根据config.m4里的信息生成了许多编译php扩展必须的文件，比如makefiles等。

第二部：运行./configure脚本，然后执行make; make test即可。如果没有错误，那么在module文件夹下面便会生成扩展的目标文件 myfirst.so，这里由于之前我们在配置文件里写申明的是动态扩展，所以会被编译成动态库。

现在，先让我们执行一下PHP源码根目录下的./buildconf —force，再执行./configure --help命令。会发现myfirst扩展的信息已经出现了。

为了使PHP能够找到需要的扩展文件，我们需要把编译好的so文件复制到PHP的扩展目录下，并在php.ini中配置：

extension_dir=/usr/local/lib/php/modules/

extension=myfirst.so

这样php就会在每次启动的时候自动加载我们的扩展了。

4.扩展功能函数编写

前面我们已经生成好了一份扩展框架，但它是没有什么实际作用的，我们还需要编写具体的功能函数。

#definePHP_FUNCTION ZEND_FUNCTION

#defineZEND_FUNCTION(name) ZEND_NAMED_FUNCTION(ZEND_FN(name))

#defineZEND_NAMED_FUNCTION(name) void name(INTERNAL_FUNCTION_PARAMETERS)

#define ZEND_FN(name) zif_##name

其中zif是zend internal function 的意思，zif前缀是可供PHP语言调用的函数在C语言中的函数名称前缀。

```
ZEND_FUNCTION(myfirst_hello)
{
    php_printf("HelloWorld!\n");
}
```

上面的函数在C语言中宏展开后是这样的:

```
voidzif_myfirst_hello(INTERNAL_FUNCTION_PARAMETERS)
{
    php_printf("HelloWorld!\n");
}
```

函数的功能已经实现了，但是还不能在程序中调用，因为这个函数还没有在扩展模块中注册。现在看下扩展中zend_module_entry

myfirst_module_entry（它是联系C扩展与PHP语言的重要纽带）中/*Functions*/的值为NULL，这是之前还没有编写函数。现在我们可以将编写的函数赋值给它了，这个值需要是zend_function_entry[]类型:

```
static zend_function_entrymyfirst_functions[] = {
     ZEND_FE(myfirst_hello, NULL)
    { NULL, NULL,NULL }
};
```

其 中最后的{NULL,NULL,NULL}是固定不变的。ZEND_FE()宏函数是对myfirst_hello函数的一个声明，如果有多个函数，可以 直接以类似的形式添加到{NULL,NULL,NULL}之前，注意每个之间不需要加逗号。确保一切无误后，我们替换掉 zend_module_entry里的原有成员，现在应该是这样的：

```
ZEND_FUNCTION(myfirst_hello)
{
    php_printf("HelloWorld!\n");
}

static zend_function_entrymyfirst_functions[] = {
    ZEND_FE(myfirst_hello,       NULL)
    { NULL, NULL,NULL }
};

zend_module_entrymyfirst_module_entry = {
#if ZEND_MODULE_API_NO >= 20010901
     STANDARD_MODULE_HEADER,
#endif
    "myfirst",//扩展名称。
    myfirst_functions,/* Functions */
    NULL, /*MINIT */
    NULL, /*MSHUTDOWN */
    NULL, /*RINIT */
    NULL, /*RSHUTDOWN */
```

```
    NULL, /*MINFO */
#if ZEND_MODULE_API_NO >= 20010901
    "2.1",//这个地方是我们扩展的版本
#endif
    STANDARD_MODULE_PROPERTIES
};
```

这样我们就完成扩展的一个简单功能，然后再重新configure 、make、make test，并复制.so文件到extension dir目录。

最后写一个脚本在命令行测试，应该可以输出helloworld了。

```
<?php
myfirst_hello();
?>
```

原文链接：http://blog.csdn.net/heiyeshuwu/article/details/40041601

为什么唱吧iOS 6.0选择了Mantle

作者：王珂

最近唱吧iOS的6.0版本已经成功上线了。18人月的投入，2500个commit，几十万行的代码修改。唱吧iOS已经从内至外焕然一新，感谢一起并肩作战的小伙伴们。

6.0一个很重大的修改就是基于Mantle重建(新建)了Model层。这里不对Mantle作更多介绍，只分享一下使用Mantle的决策及执行过程。

我们遇到的问题

唱吧是一款上线2年多的App，产品形态的演进和迭代非常快。因此不可避免的遗留了各种问题：

- Model层不健全，没有统一的结构，不同工程师做法差异很大；多数是哑类型，且没有统一的序列化机制
- 业务逻辑冗余、分散、不一致
- 模块划分随意，依赖关系混乱，维护困难
- NSDictionary作为承载业务的数据类型在各处出现(sqlite, Model object, API, Notification, web, OpenURL etc.)，参数和值的正确性完全没有编译器检查，字符串很容易写错，风险延后至运行时，易产生低级bug
- 基本没有文档和注释(结合上一点，不挂debugger很难读懂代码)
- 几百个API，业务复杂，变动快，重构难；同一个API请求可能有重复和不一致
- API的一些参数和返回值，同一个参数/返回值 可能存在类型差异；由于API需要向前兼容，修改API有成本

除此之外，还有其他工程上的约束：

- 不能影响现有的API，所有的事情只限于iOS端的修改
- 代码即文档，因为没有精力维护文档
- 对不同Model的持久化方式作迁移
- 避免写大段枯燥的Model的序列化/反序列化代码
- 没有时间造出足够成熟、健壮可重用的组件及撰写文档

上述的问题都是长期存在且需要解决的，否则严重影响开发效率及代码质量。11年的时候我还在做社交游戏的时候，设计并实现了一套简单的基于Objective-C Runtime的数值表Model结构及转换工具(Model<=>csv)供数值策划使用。但想写出一套成熟的方案还是有一些距离，而且也 没有资源和时间作维护、测试和文档。

顺着这个思路找到了JSONModel和Mantle，前者刚刚1.0，后者在Github for Mac中广泛使用且社区更成熟(甚至Slack上有channel)，所以成为了更好的选择。

事实也证明这个选择是对的，6.0上线后，crash率比之前的版本有显示的降低，并且Mantle相关的crash占总crash的比率不到3%，大可以直接用在大型的产品上。

除了成熟稳定，Mantle基本解决了我们遇到了的所有问题。下面具体介绍一些通用性Mantle使用经验，基本的使用方法请直接移步Mantle的README。

Property名称转换

由于API使用的开发语言与iOS所使用的Objective-C是截然不同的，所以可能将一些保留关键字作为property的名称(如id)， 或者不小心override掉基类的属性(如description)。还有可能API中使用了一个很糟糕的名称，或者使用了不符合Objective- C命名规范的名称，这些我们都需要作转换。

只需要实现MTLJSONSerializing protocol并在+JSONKeyPathsByPropertyKey 方法中定义好新旧名称的映射关系即可，Mantle会在序列化及反序列化时对属性名进行自动的转换。

```
+ (NSDictionary *)JSONKeyPathsByPropertyKey {
    return @{
        @"identifier": @"id",
        @"displayDiscription": @"description",
        @"thisIsANewShit": @"newShit",
        @"creativeProduct": @"copyToChina",
        @"betterPropertyName": @"m_wired_propertyName"
    }
}
```

好了很多吧？没错，只需要定义一次名称的映射关系就可以了，Mantle负责model与JSON之间的双向转换。不需要将这种逻辑写得到处都是，并且还得维护它的一致性。

Property的类型映射

iOS中处理URL使用的是NSURL类型，但JSON只支持基本的字符串，Mantle可以自动帮你转换成NSURL。

```
+ (NSValueTransformer *)URLJSONTransformer {
    return [NSValueTransformer
valueTransformerForName:MTLURLValueTransformerName];
}
```

NSValueTransformer负责在不同类型间进行双向转换，请读者研究一下Mantle的实现方式。在此前提下，留给读者一个问题(其实这是一个真实的故事，类似的故事还有很多，详见iOS应用开发之十大坑队友)：

假设我们有一个entity，名字且叫KTVConcreteEntity吧，它有一个属性名字叫entityID，类型是NSInteger。问 题来了，entityID可能在另外一个API的response中是字符串类型，在不直接修改Mantle的源码的前提下怎么搞？欢迎在下方留言讨论。

空标量异常

有的时候API的response会有空值，比如copyToChina可能不是每次都有的，JSON是这样儿的：

```
{
    "copyToChina": null
}
```

Mantle在这种情况会将newShit转换为nil，但如果是标量如NSInteger怎么办？KVC会直接raise NSInvalidArgumentException。

Mantle是基于KVC给property赋值的，KVC提供了-(void)setNilValueForKey:(NSString *)key方法，让我们为nil指定一个合理的替代值，我们来看一下此方法的解释：

Invoked by setValue:forKey: when it’s given a nil value for a scalar value (such as an int or float).
Subclasses can override this method to handle the request in some other way, such as by substituting 0 or a sentinel value for nil and invoking setValue:forKey: again or setting the variable directly. The default implementation raises an NSInvalidArgumentException.

对于标量来讲，多数情况下合理的值即为0，我们来看下代码：

```
@interface MTLModel (KTVNullableScalar)
@end

@implementation MTLModel (KTVNullableScalar)
- (void)setNilValueForKey:(NSString *)key {
    [self setValue:@0 forKey:key];  // For NSInteger/CGFloat/BOOL
}
@end
```

问题完美解决，再也不需要到处写无聊的if/else了。

其它重要特性

Mantle为我们带来的方便不胜枚举：

- 实现了NSCopying protocol，子类可以直接copy是多么爽的事情
- 实现了NSCoding protocol，跟NSUserDefaults说拜拜
- 提供了-isEqual:和-hash的默认实现，model作NSDictionary的key方便了许多
- 简单且把一件事情做好，不掺杂网络相关的操作

如此强大优雅的设计，让我不得不向Github的工程师们致敬！

写在后面

篇幅所限，只介绍了几个典型的问题，欢迎大家讨论。但如果你的App的代码规模只有几万行，或者API只有十几个，或者没有遇到我们这些遗留问

题， 我建议还是不要引入了，杀鸡用指甲刀就够了，杀不动多磨磨找准要害。Anyway，Mantle的实现和思路是值得每位iOS工程师学习和借鉴的。

原文链接：http://www.iwangke.me/2014/10/13/Why-Changba-iOS-choose-Mantle/?utm_source=tuicool

State Threads 回调终结者

作者：我的上铺叫路遥

上回写了篇《一个“蝇量级”C语言协程库》，推荐了一下Protothreads，通过coroutine模拟了用户级别的multi-threading模型，虽然本身足够“轻”，杜绝了系统开销，但这个库本身应用场合主要是内存限制的嵌入式领域，提供原生态组件太少，使用限制太多，比如依赖其它调用产生阻塞等。

这回又替大家在开源界淘了个宝，推荐一个轻量级网络应用框架State Threads（以下简称ST），总共也就3000行C代码，跟Protothreads不同在于ST针对的就是高性能可扩展服务器领域（值得一提的是Protothreads官网参考链接上第一条就是ST的官网）。在其FAQ页 面上一句引用”Perfection is achieved not when there is nothing more to add, but rather when there is nothing more to take away.”可以视为开发人员对ST源码质量的自信。

历史渊源

首先介绍一下这个库的历史渊源，从代码贡献者来看，ST不是个人作品，而是有着雄厚的商业支持和应用背景，比如服务器领域，在这里你 可以看到ST曾作为Apache的多核应用模块发布。其诞生最初是由网景（Netscape）公司的MSPR（Netscape Portable Runtime library）项目中剥离出来，后由SGI（Silicon Graphic Inc）还有Yahoo!公司（前者是主力）开发维护的独立线程库。历史版本方面，作为SourceForge上 开源项目，由2001年发布v1.0以来一直到2009年v1.9稳定版后未再变动。在平台移植方面，从Makefile的配置选项中可知ST支持多种 Unix-like平台，还有专门针对Win32的源码改写。源码例子中，提供了web server、proxy以及dns三种编程实例供参考。可以说代码质量应该是相当的稳定和可靠的。

至于许可证方面，有必要略作说明。出于历史原因，网景最初发布时选择了MPL1.1许可证，而后SGI在维护中又混进了GPLv2许可证，照理说这两种许可证是互不兼容的（MPL1.1后续版本是GPL兼容的），也就是说用双许可证打包发布理论上是非法无效的，见GNU官网上MPL兼容性一 节。但这里有值得商榷的地方，因为文中又提及，根据MPL1.1中某条款第13节，如果整段或部分代码允许采用另一许可证作为备用（alternate） 选择，比如GPL及其兼容，那么整个库的许可证就可视为GPL兼容的。如此一来所谓GPL兼容性一般解释为你不能在GPLv2的代码中混入MPL1.1， 而不是说你不能在MPL1.1代码中混入GPLv2，也就是说GPLv2在MPL1.1之后是可以接受的，事实上SGI就采用了后面的做法，尚未引起版权 上的纠纷。为此我还考证了一下FAQ上license一 节的说法，说ST既可以在MPL和GPL之间选择一种，也可以继续用双许可证，还补了一句在non-free项目使用上也没有限制，但对ST源码所做改动 必须对用户可见。在源码文件中的SGI的附加声明还解释了将ST转为GPL代码的做法，就是可以删除前面MPL的声明，否则后续用户仍可以在两者之间二选 一。个人觉得既然SGI都这样发话了，那么可解释为反之删除GPL的声明继续采用MPL也是可以接受的，如果你对双许可证承诺仍不放心的话。

基于事件驱动状态机（EDSM）

好了，下面该进入技术性话题了。前面说了ST的目标是高性能可扩展，其技术特征一言以蔽之就是

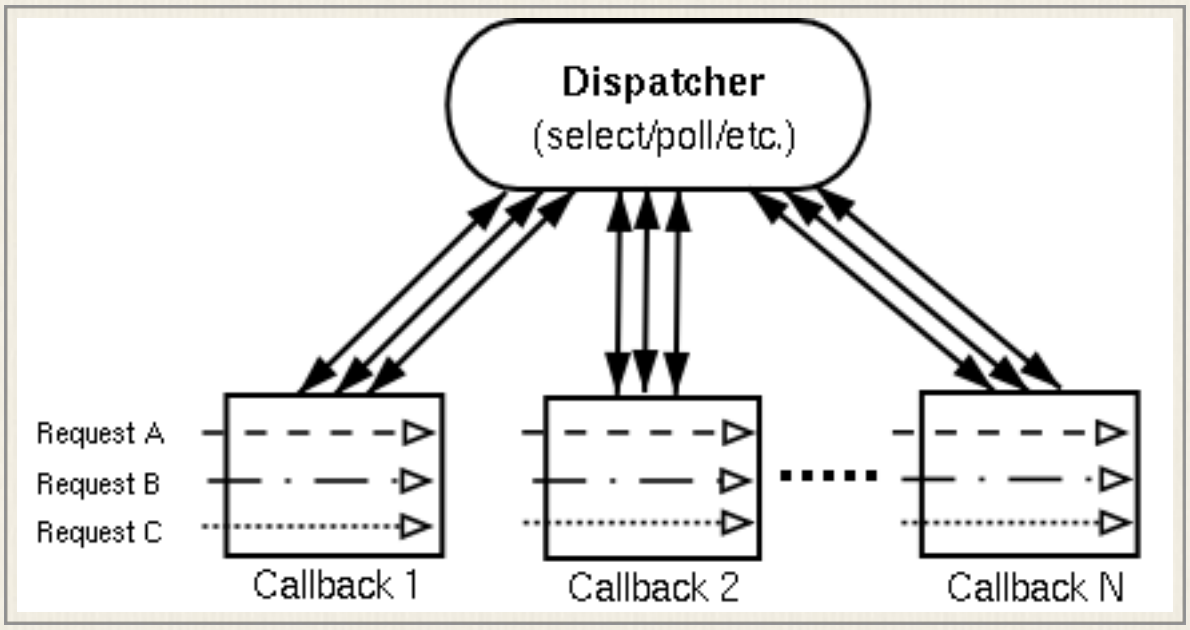
“It combines the simplicity of the multi-threaded programming paradigm, in which one thread supports each simultaneous connection, with the performance and scalability of an event-driven state machine (EDSM) architecture.”

我们先来纵向比较ST与传统的EDSM区别，再来横向比较与其它线程库（比如Pthread）的区别（注：以下图片全部来自State Threads Library FAQ）。

传统EDSM最常见的方式就是I/O 事件的异步回调。基本上都会有一个叫做dispatcher的单线程主循环 （又叫event loop），用户通过向dispatcher注册回调函数（又叫event handler）来实现异步通知，从而不必在原地空耗资源干等，在dispatcher主循环中通过select()/poll()系统调用来等待各种I

/O事件的发生，当内核检测到事件触发并且数据可达或可用时，select()/poll()会返回从而使dispatcher调用相应的回调函数来对处 理用户的请求。所以异步回调与其说是通知，不如说用委托更恰当。

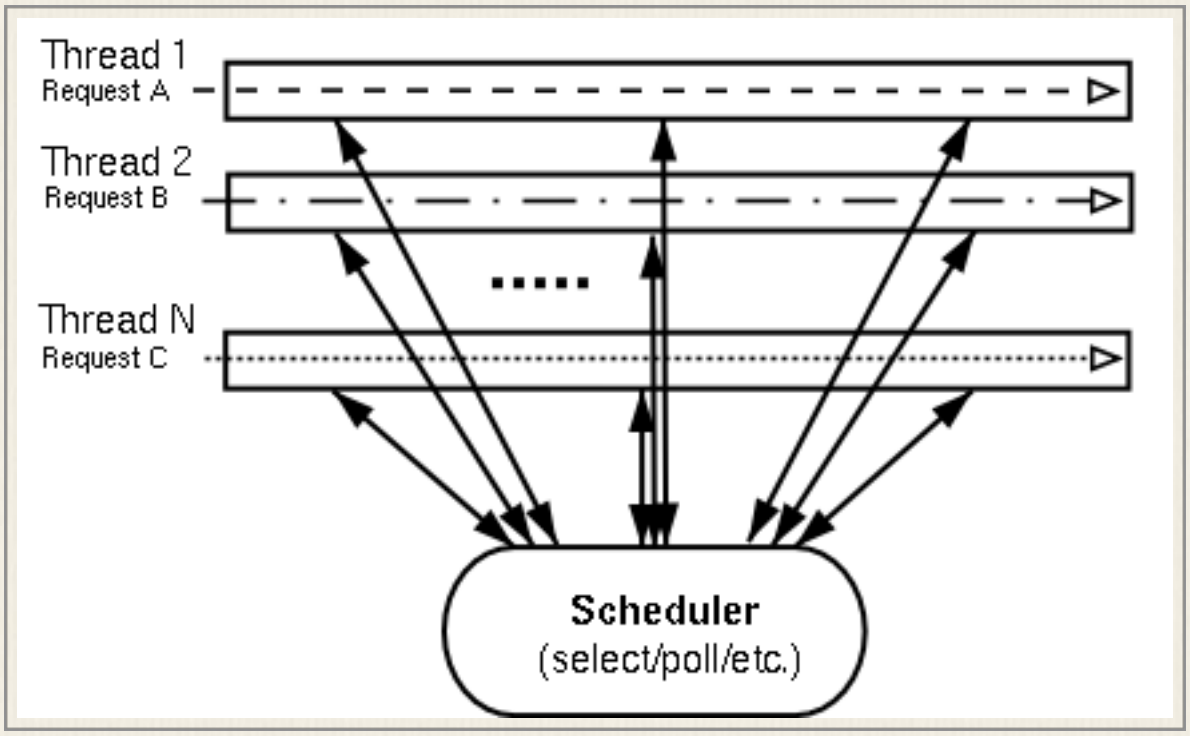
整个过程都是单线程的。这种处理本质上就是将一堆互不相交（disjoint）的回调实现同步控制，就像串联在一个顺序链表上。见 图1，黑色的双箭头表示I/O事件复用，回调是个筐，里面装着对各种请求的处理（当然不是每个请求都有回调，一个请求也可以对应不同的回调），每个回调被 串联起来由dispatcher激活。这里请求等价于thread的概念（不是操作系统的线程），只不过“上下文切换”（context switch）发生在每个回调结束之时（假设不同请求对应不同回调），注册下一个回调以待事件触发时恢复其它请求的处理。至于dispatcher的执行 状态（execute state）可作为回调函数的参数保存和传递。



异步回调的缺陷在于难以实现和扩展，虽然已经有libevent这样的通用库，以及其它actor/reacotor的设计模式及其框架，但正如Dean Gaudet（Apache开发者）所说：“其内在的复杂性——将线性思维分解成一堆回调的负担（breaking up linear thought into a bucketload of callbacks）——仍然存在”。从上图可见，回调之间请求例程不是连续的，比如回调之间的切换会打断部分请求，又比如有新的请求需要重新注册。

ST本质上仍然是基于EDSM模型，但旨在取代传统的异步回调方式。ST将请求抽象为thread概念以更接近 自然编程模式（所谓的linear thought吧，就像操作系统的线程之间切换那样自然）。ST的调度器（scheduler）

对于用户来说是透明的，不像dispatcher那种将执 行状态（execute state）暴露给回调方式。每个thread的现场环境可以保存在栈上（一段连续的大小确定的内存空间），由C的运行环境管理。从图2看到，ST的threads可以并发地线性地处理I/O事件，模型比异步回调简单得多。



这里稍微解释一下ST调度工作原理，ST运行环境维护了四种队列，分别是IOQ、RUNQ、SLEEPQ以及ZOMBIEQ，当每个thread处于不同队列中对应不同的状态（ST顾名思义所谓thread状态机）。比 如polling请求的时候，当前thread就加入IOQ表示等待事件（如果有timeout同时会被放到SLEEPQ中），当事件触发 时，thread就从IOQ（如果有timeout同时会从SLEEPQ）移除并转移到RUNQ等待被调度，成为当前的running thread，相当于操作系统的就绪队列，跟传统EDSM对应起来就是注册回调以及激活回调。再比如模拟同步控制wait/sleep/lock的时候， 当前thread会被放入SLEEPQ，直到被唤醒或者超时再次进入RUNQ以待调度。

ST的调度具备性能与内存双重优点：在性能上，ST实现自己的setjmp/longjmp来模拟调度，无任何系统开销，并且context（就是jmp_buf）针对不同平台和架构用底层语言实现的，可移植性媲美libc。下面放一段代码解释一下调度实现：

```
/*
 * Switch away from the current thread context by saving its state
 * and calling the thread scheduler
 */
#define _ST_SWITCH_CONTEXT(_thread)       \
    ST_BEGIN_MACRO                        \
    if (!MD_SETJMP((_thread)->context)) { \
      _st_vp_schedule();                  \
    }                                     \
    ST_END_MACRO

/*
    * Restore a thread context that was saved by _ST_SWITCH_CON-
TEXT
    * or initialized by _ST_INIT_CONTEXT
    */
   #define _ST_RESTORE_CONTEXT(_thread)   \
      ST_BEGIN_MACRO                      \
      _ST_SET_CURRENT_THREAD(_thread);    \
      MD_LONGJMP((_thread)->context, 1); \
      ST_END_MACRO

   void _st_vp_schedule(void)
   {
```

```
_st_thread_t *thread;

if (_ST_RUNQ.next != &_ST_RUNQ) {
    /* Pull thread off of the run queue */
    thread = _ST_THREAD_PTR(_ST_RUNQ.next);
    _ST_DEL_RUNQ(thread);
} else {
    /* If there are no threads to run, switch to the idle thread */
    thread = _st_this_vp.idle_thread;
}
ST_ASSERT(thread->state == _ST_ST_RUNNABLE);

    /* Resume the thread */
    thread->state = _ST_ST_RUNNING;
    _ST_RESTORE_CONTEXT(thread);
}
```

如果你熟悉setjmp/longjmp的用法，你就知道当前thread在调用MD_SETJMP将现场上下文保存在jmp_buf中并返回返回 0，然后自己调用_st_vp_schedule()将自己调度出去。调度器先从RUNQ上找，如果队列为空就找idle thread，这是在整个ST初始化时创建的一个特殊thread，然后将当前线程设为自己，再调用MD_LONGJMP切换到其上次调用MD_SETJMP的地方，从thread->context恢复现场并返回1，该thread就接着往下执行了。整个过程就同EDSM一样发生在操作系统单线程下，所以没有任何系统开销与阻塞。

其实真正的阻塞是发生在等待I/O事件复用上，也就是select()/poll()，这是整个ST唯一的系统调用。ST 当前的状态是，整个环境处于空闲状态，所有threads的请求处理都已经完成，也就是RUNQ为空。这时在_st_idle_thread_start 维护了一个主循环（类似于event loop），主要负责三种任务：1.对IOQ所有thread进行I/O复用检测；2.对SLEEPQ进行超时检查；3.将idle thread调度出去，代码如下：

```
void *_st_idle_thread_start(void *arg)
{
    _st_thread_t *me = _ST_CURRENT_THREAD();

    while (_st_active_count > 0) {
        /* Idle vp till I/O is ready or the smallest timeout expired */
        _ST_VP_IDLE();

        /* Check sleep queue for expired threads */
        _st_vp_check_clock();

        me->state = _ST_ST_RUNNABLE;
        _ST_SWITCH_CONTEXT(me);
    }

    /* No more threads */
    exit(0);
```

```
    /* NOTREACHED */
    return NULL;
}
```

这里的me就是idle thread，因为_st_idle_thread_start就是创建idle thread的启动点，每从上次_ST_SWITCH_CONTEXT()切换回来的时候，接着在_ST_VP_IDLE()里轮询I/O事件的发生，一 旦检测到发生了别的thread事件或者SLEEPQ里面发生超时，再用_ST_SWITCH_CONTEXT()把自己切换出去，如果此时RUNQ中非 空的话就切换到队列第一个thread。这里主循环是不会退出的。

在内存方面，ST的执行状态作为局部变量保存在栈上，而不是像回调需要动态分配，用户可能分别这样使用thread模式和callback模式：

```
/* thread land */
int foo()
{
    int local1;
    int local2;
    do_some_io();
}

/* callback land */
struct foo_data {
    int local1;
    int local2;
};
```

```
void foo_cb(void *arg)
{
    struct foo_data *locals = arg;
    ...
}

void foo()
{
    struct foo_data *locals = malloc(sizeof(struct foo_data));
    register(foo_cb, locals);
}
```

基于Mult-Threading范式

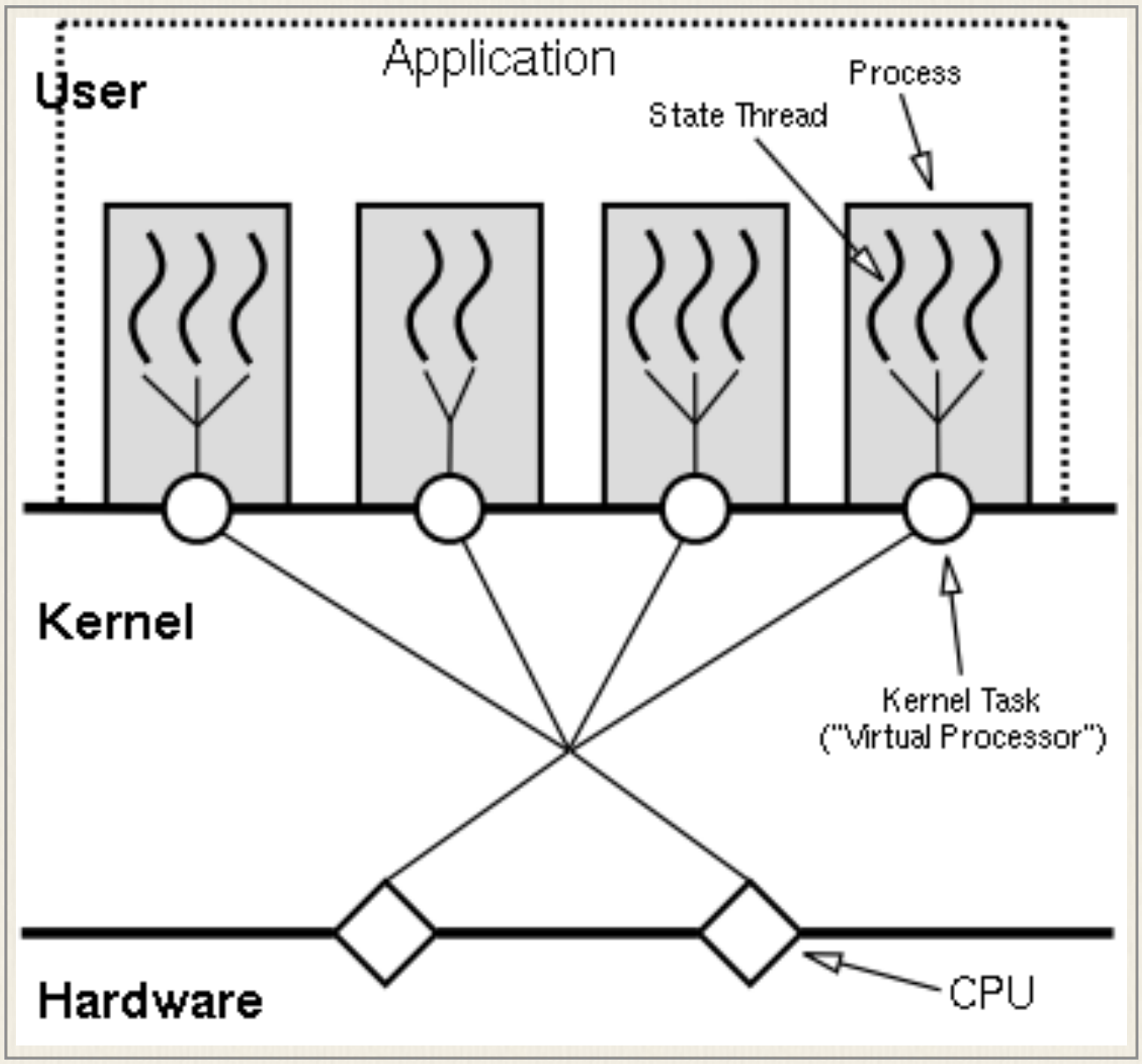
同样基于multi-threading 编程范式，ST同其它线程库又有和有点呢？比如Posix Thread（以下简称PThread）是个通用的线程库，它是将用户级线程（thread）同内核执行对象（kernel execution entity，有些书又叫light-weight processes）做了1:1或m:n映射，从而实现multi-threading模式。而ST是单线程（n:1映射），它的thread实际上就是协程（coroutine）。通常的网络应用上，多线程范式绕不开操作系统，但在某些特定的服务器领域，线程间的共享资源会带来额外复杂度，锁、竞态、并发、文件句柄、全局变量、管道、信号等，面对这些Pthread的灵活性会大打折扣。而ST的调度是精确的，它只会在明确的I/O和同步函数调用点上发生上下文切换，这正是协程的特性，如此一来ST就不需要互斥保护了，进而也可以放心使用任何静态变量和不可重入库函数了（这在同样作为协程的Protothreads里是不允许的，因为那是stack-less的，无法保存上下文），极大的简化了编程和调试同时增加了性能。

对于同样用户级线程如GNU Pth和MIT Phread比起来呢？有两点，一是ST的thread是无优先级的非抢占式调度，也就是说ST基于EDSM的，每个thread都是事件或数据驱动，迟早会把自己调度出去，而且调度点是明确的，并非按时间片来的，从而简化了thread管理；二是ST会忽略所有信号处理， 在_st_io_init中会把sigact.sa_handler设为SIG_IGN，这样做是因为将thread资源最小化，避免了signal mask及其系统调用（在ucontext上是避免不了的）。但这并不意味着ST就不能处理信号，实际上ST建议将信号写入pipe的方式转化为普通 I/O事件处理，示例详见这里。

这里顺便说一句，C语言实现的协程据我所知只有三种方式：Protothread为代表利用switch-case语义跳转，以ST为代表不依赖libc的setjmp/longjmp上下文切换，以及依赖glibc的ucontext接口（云风的coroutine）。第一种最轻，但受限最大，第三种耗资源性能慢（陈皓注：glibc的ucontext接口的实现中有一个和信号有关的系统调用，所以会慢，估计在一些情况下会比pthread还慢），目前看来ST是最好使的。

基于多核环境

下面来聊聊ST在多核环境下的应用。服务器领域多核的优势在于实现了物理上真正的并发，所以如何充分利用系统优势也是线程库的一大难点。这对ST来 说也许正是它的拿手好戏，前面提及ST曾作为Apache的多核引擎模块发布。这里要补充一下前面漏掉的ST的一个重要概念——虚拟处理器（virtual processor，简称vp），见图3，多个cpu通过内核的SMP模拟出多个“核”（core），一个core对应一个内核任务（kernel task），同时对应一个用户进程（process），一个process对应ST的一个vp，每个vp下就是ST的thread（是协程不是线程），结 合前面所述，vp初始化先创建idle thread，然后根据I/O事件驱动其它threads，这就是ST的多核架构。



这里要指出的是，ST只负责自身thread调度，进程管理是应用程序的事情，也就是说由用户来决定fork多少进程，每个进程分配多少资源，如何进行IPC等。这种架构的好处就是每个vp有自己独立的空间，避免了资源同步竞态（比如杜绝了多进程里的多线程这样混乱的模型）。我们知道这种基于进程的架构是非常健壮的，一个进程奔溃不会影响到其它进程，同时充分利用多核硬件的高并发。同时对于具体逻辑业务使用vp里的thread处理，这是基于EDSM的，如此一来做到了逻辑业务与内核执行对象之间的解耦，没必要因为1K个连接去创建1K的进程。这就是ST的扩展性和灵活性。

使用限制

ST的主要限制在于，应用程序所有I/O操作必须使用ST提供的API，因为只有这样thread才能被调度器管理，并且避免阻塞。

另一个限制在于thread调试，这本身不容易，好在v1.9的ST提供了DEBUG参数，使用TREADQ以及 _st_iterate_threads接口检测thread调度情况，用户还可自定义_st_show_thread_stack接口dump每个 thread的栈，在GDB使能_st_iterate_threads_flag变量，这些都在Readme中对调试方法有具体说明。按下不表。

总结

这篇文章写得有点短了，主要是通过对比来介绍ST的，其实还有大段原理可以讲，大段源码以及实战用例可以贴，但这一下子又写不过来，ST还是有点技术含量的。说白了，ST的核心思想就是利用multi-threading的简单优雅范式胜过传统异步回调的复杂晦涩实现，又利用EDSM的性能和解耦架构避免了multi-threading在系统上的开销和暗礁。学习ST告诉我们一个道理：未来技术的趋势永远都是融合的。

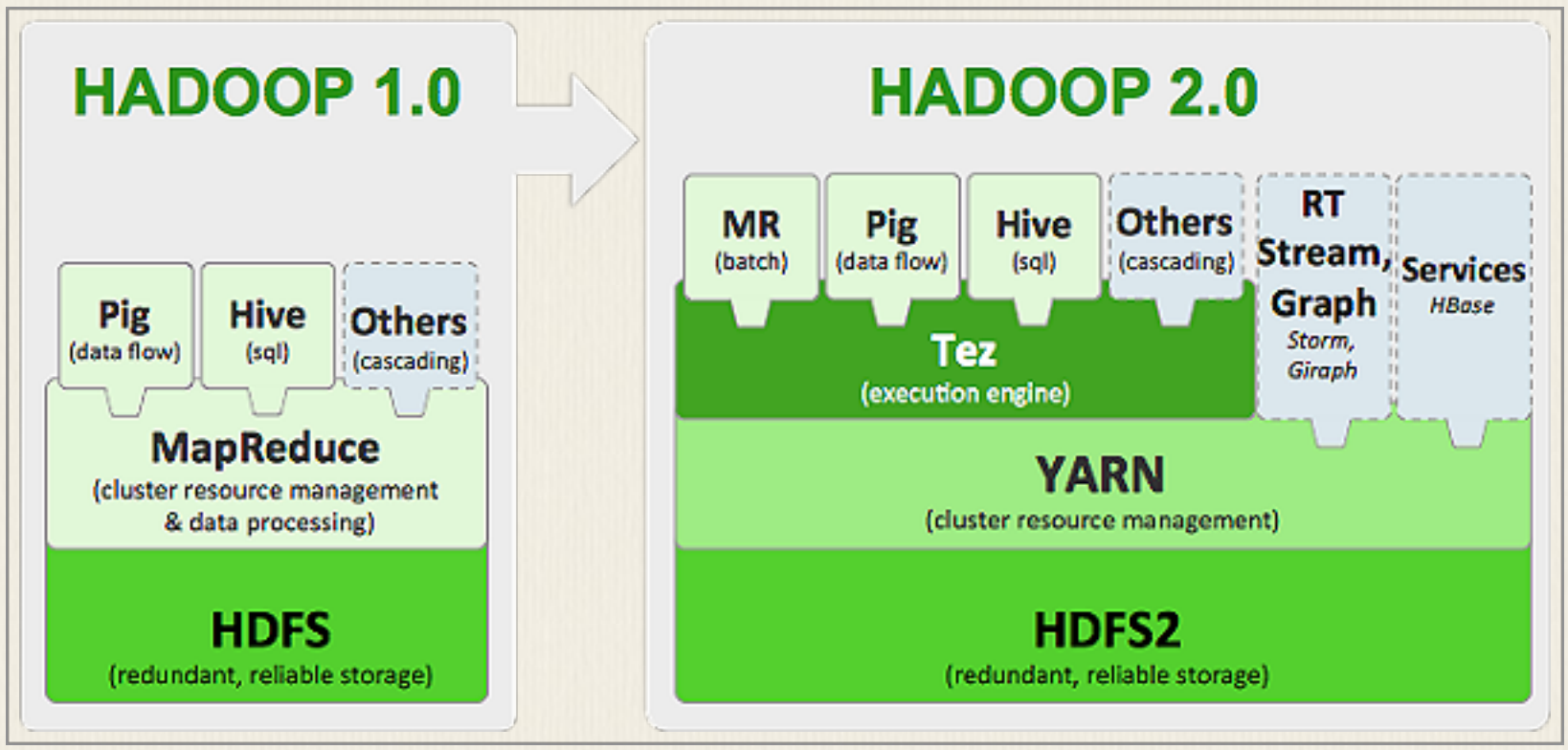
原文链接：http://coolshell.cn/articles/12012.html

Apache Tez是什么？

译者：孙镜涛

Tez是Apache最新的支持DAG作业的开源计算框架，它可以将多个有依赖的作业转换为一个作业从而大幅提升DAG作业的性能。Tez并不直接 面向最终用户——事实上它允许开发者为最终用户构建性能更快、扩展性更好的应用程序。Hadoop传统上是一个大量数据批处理平台。但是，有很多用例需要 近乎实时的查询处理性能。还有一些工作则不太适合MapReduce，例如机器学习。Tez的目的就是帮助Hadoop处理这些用例场景。

Tez项目的目标是支持高度定制化，这样它就能够满足各种用例的需要，让人们不必借助其他的外部方式就能完成自己的工作，如果 Hive和 Pig 这样的项目使用Tez而不是MapReduce作为其数据处理的骨干，那么将会显著提升它们的响应时间。Tez构建在YARN之上，后者是Hadoop所使用的新资源管理框架。



设计哲学

Tez产生的主要原因是绕开MapReduce所施加的限制。除了必须要编写Mapper和Reducer的限制之外，强制让所有类型的计算都满足 这一范例还有效率低下的问题——例如使用HDFS存储多个MR作业之间的临时数据，这是一个负载。在Hive中，查询需要对不相关的key进行多次 shuffle操作的场景非常普遍，例如join - grp by - window function - order by。

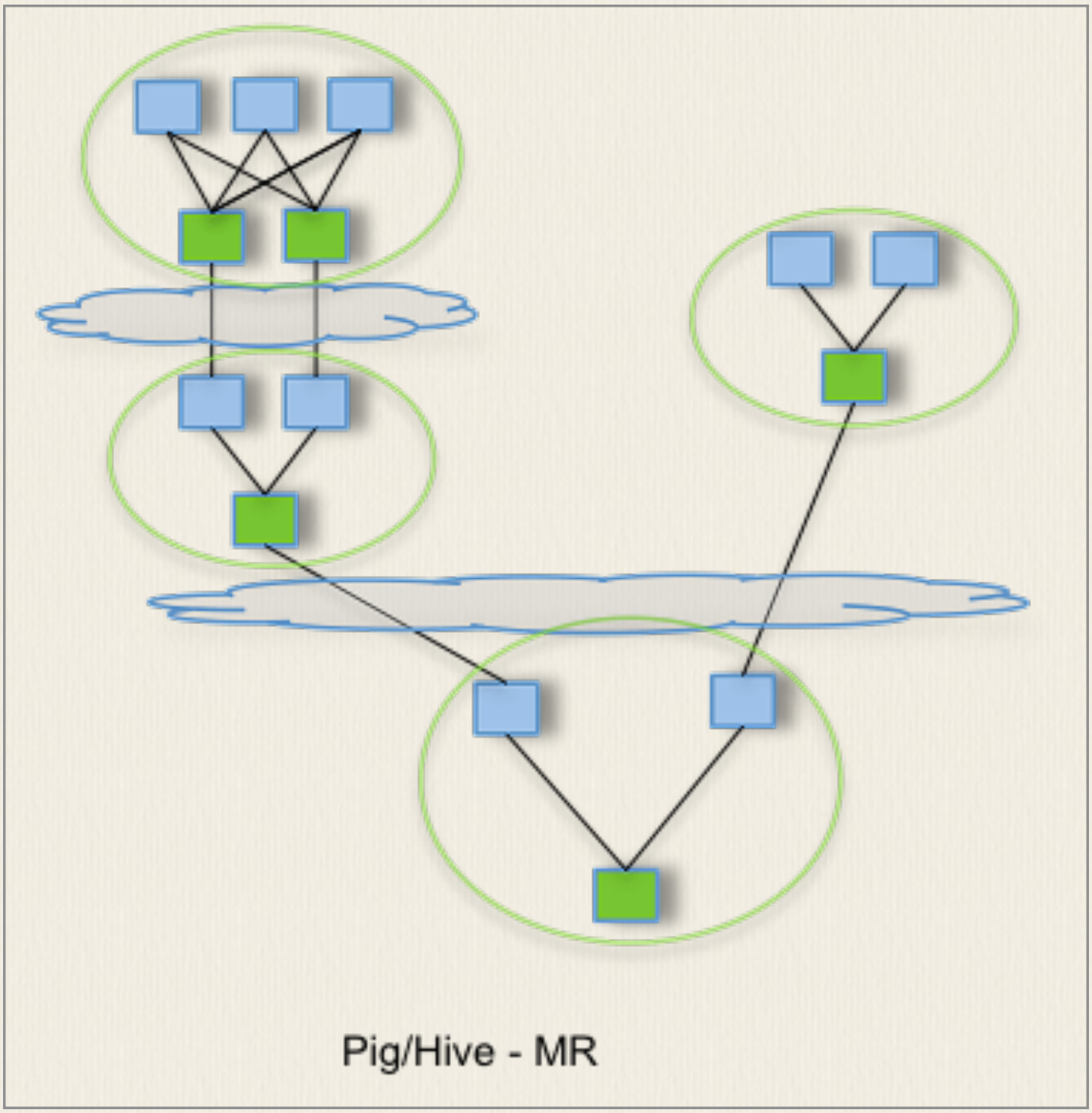
ez设计哲学里面的关键元素包括：

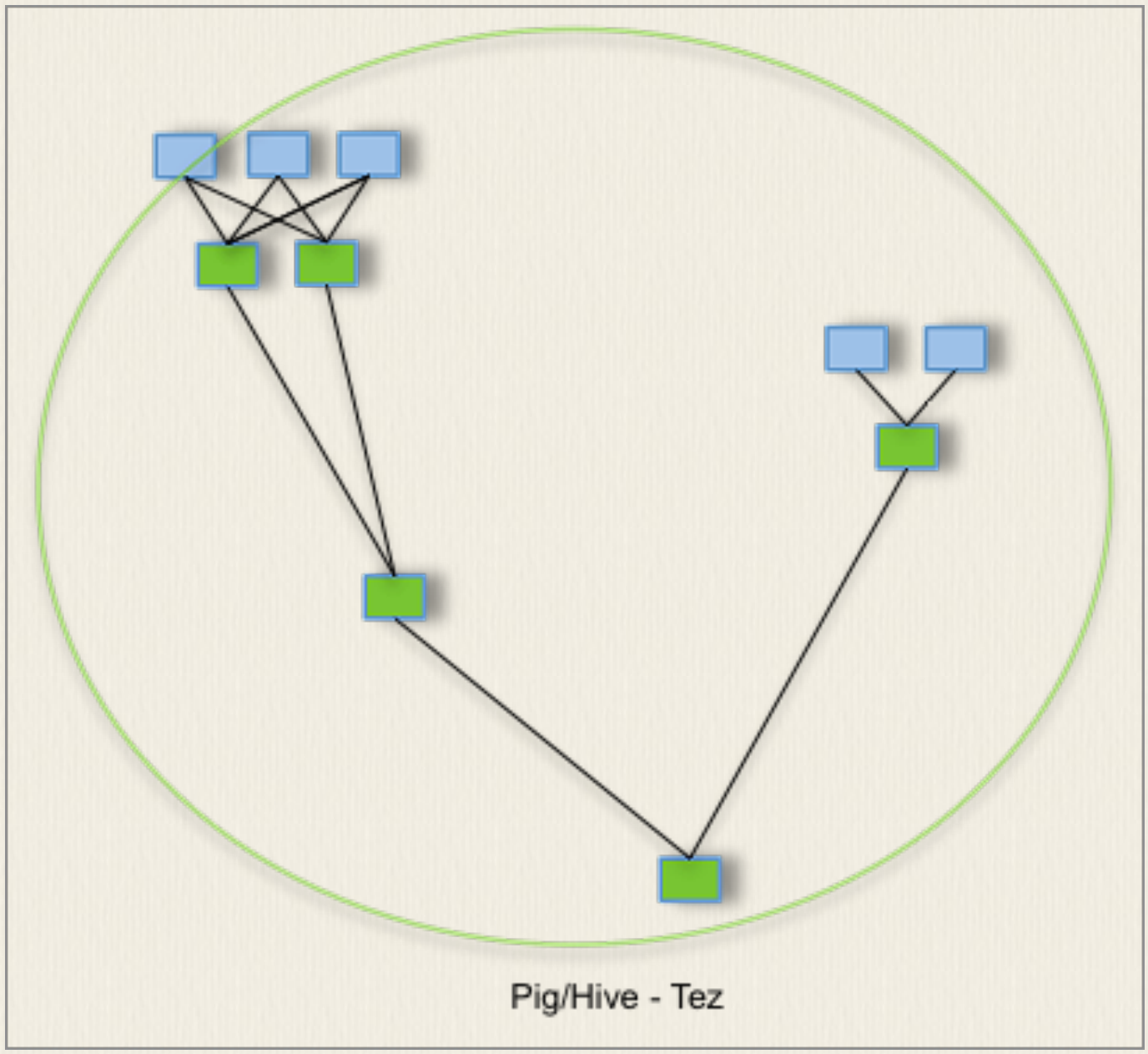
- 允许开发人员（也包括最终用户）以最有效的方式做他们想做的事情
- 更好的执行性能

Tez之所以能够实现这些目标依赖于以下内容：

- 具有表现力的数据流API——Tez团队希望通过一套富有表现力的数据流定义API让用户能够描述他们所要运行计算的有向无环图 （DAG）。为了达到这个目的，Tez实现了一个结构化类型的API，你可以在其中添加所有的处理器和边，并可视化实际构建的图形。
- 灵活的输入—处理器—输出（Input-Processor-Output）运行时模型——可以通过连接不同的输入、处理器和输出动态地构建运行时执行器。
- 数据类型无关性——仅关心数据的移动，不关心数据格式（键值对、面向元组的格式等）。
- 动态图重新配置
- 简单地部署——Tez完全是一个客户端应用程序，它利用了YARN的本地资源和分布式缓存。就Tez的使用而言，你不需要在自己的集群上部署任何内容，仅需要将相关的Tez类库上传到HDFS上，然后使用Tez客户端提交这些类库即可。 你甚至可以在你的集群上放置两份类库。一份用于产品环境，它使用稳定版本供所有的生产任务使用；另一份使用最新版本，供用户体验。这两份类库相互独立，互不影响。
- Tez能够运行任意MR任务，不需要做任何改动。这样能够让那些现在依赖于MR的工具实现分布迁移。

接下来让我们详细地探索一下这些表现力丰富的数据流API——看看我们可以使用它们做些什么？例如，你可以使用MRR模式而不是使用多个MapReduce任务，这样一个单独的map就可以有多个reduce阶段；并且这样做数据流可以在不同的处理器之间流转，不需要把任何内容写入HDFS（将会被写入磁盘，但这仅仅是为了设置检查点），与之前相比这种方式性能提升显著。下面的图表阐述了这个过程：





第一个图表展示的流程包含多个MR任务，每个任务都将中间结果存储到HDFS上——前一个步骤中的reducer为下一个步骤中的mapper提供数据。第二个图表展示了使用Tez时的流程，仅在一个任务中就能完成同样的处理过程，任务之间不需要访问HDFS。

Tez的灵活性意味着你需要付出比MapReduce更多的努力才能使用它，你需要学习更多的API，需要实现更多的处理逻辑。但是这还好，毕竟它和MapReduce一样并不是一个面向最终用户的应用程序，其目的是让开发人员基于它构建供最终用户使用的应用程序。

以上内容是对Tez的概述及其目标的描述，下面就让我们看看它实际的API。

Tez API

Tez API包括以下几个组件：

- 有向无环图（DAG）——定义整体任务。一个DAG对象对应一个任务。
- 节点（Vertex）——定义用户逻辑以及执行用户逻辑所需的资源和环境。一个节点对应任务中的一个步骤。
- 边（Edge）——定义生产者和消费者节点之间的连接。 边需要分配属性，对Tez而言这些属性是必须的，有了它们才能在运行时将逻辑图展开为能够在集群上并行执行的物理任务集合。下面是一些这样的属性：
  - 数据移动属性，定义了数据如何从一个生产者移动到一个消费者。
  - 调度（Scheduling）属性（顺序或者并行），帮助我们定义生产者和消费者任务之间应该在什么时候进行调度。
  - 数据源属性（持久的，可靠的或者暂时的），定义任务输出内容的生命周期或者持久性，让我们能够决定何时终止。

如果你想查看一个API的使用示例，对这些属性的详细介绍，以及运行时如何展开逻辑图，那么可以看看Hortonworks提供的这篇文章。

运行时API基于输入—处理器—输出模型， 借助于该模型所有的输入和输出都是可插拔的。为了方便，Tez使用了一个基于事件的模型，目的是为了让任务和系统之间、组件和组件之间能够通信。事件用于 将信息（例如任务失败信息）传递给所需的组件，将输出的数据流（例如生成的数据位置信息）传送给输入，以及在运行时对DAG执行计划做出改变等。

Tez还提供了各种开箱即用的输入和输出处理器。

这些富有表现力的API能够让更高级语言（例如Hive）的编写者很优雅地将自己的查询转换成Tez任务。

Tez调度程序

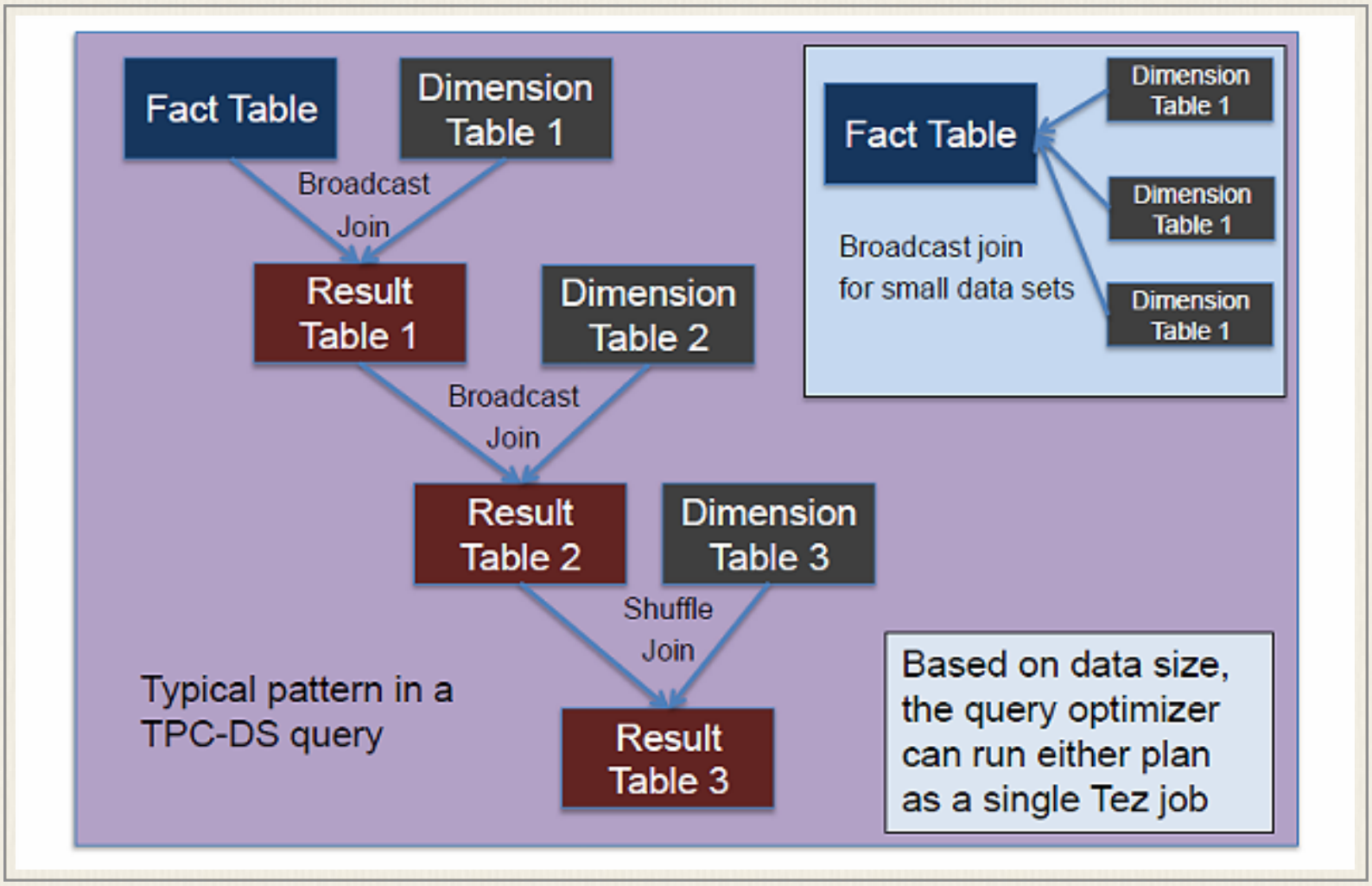
在决定如何分配任务的时候，Tez调度程序考虑了很多方面，包括：任务位置需求、容器的兼容性、集群可利用资源的总量、等待任务请求的优先级、自动 并行化、释放应用程序不再使用的资源（因为对它而言数据并不

是本地的）等。它还维护着一个使用共享注册对象的预热JVM连接池。应用程序可以选择使用这些 共享注册对象存储不同类型的预计算信息，这样之后再进行处理的时候就能重用它们而不需要重新计算了，同时这些共享的连接集合及容器池资源也能非常快地运行 任务。

如果你想了解更多与容器重利用相关的信息，那么可以查看这里。

扩展性

总体来看，Tez为开发人员提供了丰富的扩展性以便于让他们能够应对复杂的处理逻辑。这可以通过示例“Hive是如何使用Tez的”来说明。



让我们看看这个经典的TPC-DS查询模式，在该模式中你需要将多个维度表与一个事实表连接到一起。大部分优化器和查询系统都能完成该图右上角部分 所描述的场景：如果维度表较小，那么可以将所有的维度表与较大的事实表进行广播连接，这种情况下你可以在Tez上完成同样的事情。

但是如果这些广播包含用户自定义的、计算成本高昂的函数呢？此时，你不可能都用这种方式实现。这就需要你将自己的任务分割成不同的阶段，

正如该图左边的拓扑图所展示的方法。第一个维度表与事实表进行广播连接，连接的结果再与第二个维度表进行广播连接。

第三个维度表不再进行广播连接，因为它太大了。你可以选择使用shuffle连接，Tez能够非常有效地导航拓扑。

使用Tez完成这种类型的Hive查询的好处包括：

- 它为你提供了全面的DAG支持，同时会自动地在集群上完成大量的工作，因而它能够充分利用集群的并行能力；正如上面所介绍的，这意味着在多个MR任务之间不需要从HDFS上读/写数据，通过一个单独的Tez任务就能完成所有的计算。
- 它提供了会话和可重用的容器，因此延迟低，能够尽可能地避免重组。

使用新的Tez引擎执行这个特殊的Hive查询性能提升将超过100%。

路线图

- 更加丰富的DAG支持。例如，Samza是否能够使用Tez作为其底层支撑然后在这上面构建应用程序？为了让Tez能够处理Samza的核心调 度和流式需求开发团队需要做一些支持。Tez团队将探索如何在我们的DAG中使用这些类型的连接模式。他们还想提供更好的容错支持，更加有效地数据传输， 从而进一步优化性能，并且改善会话性能。
- 考虑到这些DAG的复杂度无法确定，需要提供很多自动化的工具来帮助用户理解他们的性能瓶颈。

总结

Tez是一个支持DAG作业的分布式执行框架。它能够轻而易举地映射到更高级的声明式语言，例如Hive、Pig、Cascading等。它拥有一 个高度可定制的执行架构，因而我们能够在运行时根据与数据和资源相关的实时信息完成动态性能优化。框架本身会自动地决定很多棘手问题，让它能够顺利地正确 运行。

使用Tez，你能够得到良好的性能和开箱即用的效率。Tez的目标是解决Hadoop数据处理领域所面对的一些问题，包括延迟以及执行的复杂性等。Tez是一个开源的项目，并且已经被Hive和Pig使用。

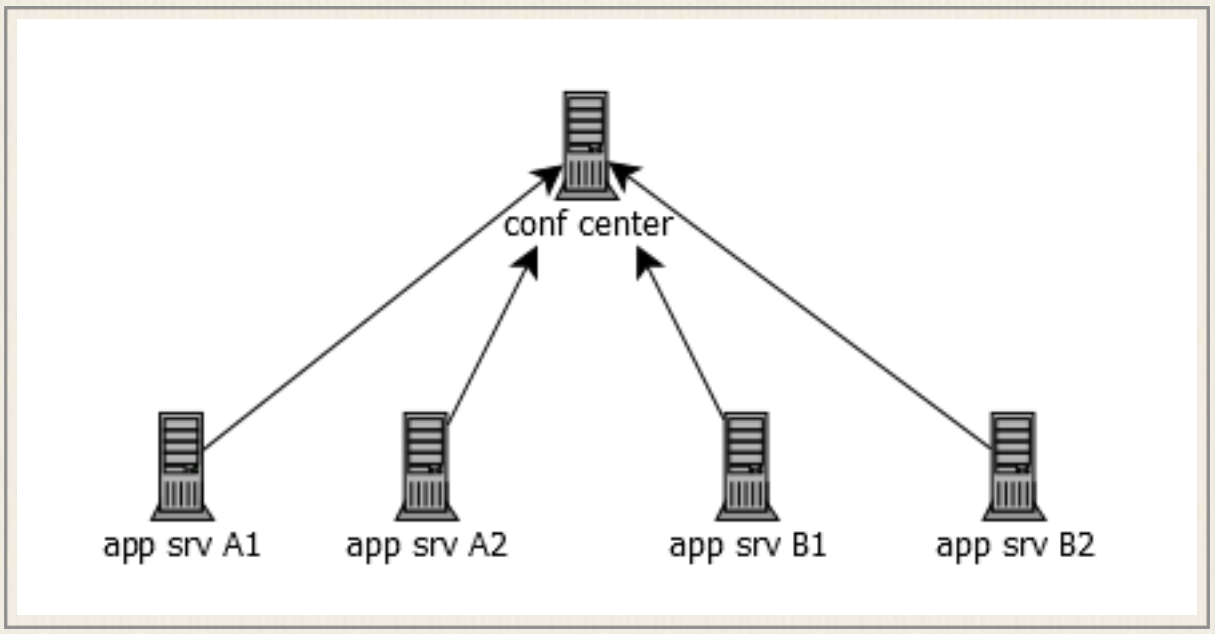
译文链接：http://www.infoq.com/cn/articles/apache-tez-saha-murthy

原文链接：http://www.infoq.com/articles/apache-tez-saha-murthy

淘宝分布式配置管理服务Diamond

作者：kevinlynx

在一个分布式环境中，同类型的服务往往会部署很多实例。这些实例使用了一些配置，为了更好地维护这些配置就产生了配置管理服务。通过这个服务可以轻松地管理这些应用服务的配置问题。应用场景可概括为：

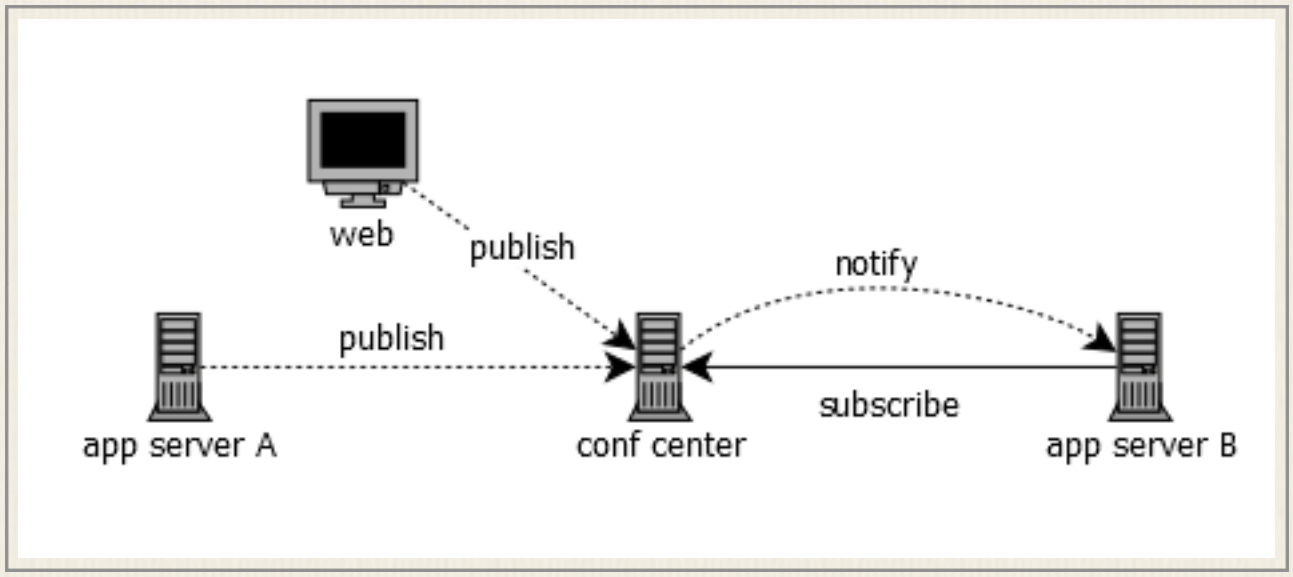


zookeeper的一种应用就是分布式配置管理(基于ZooKeeper的配置信息存储方案的设计与实现)。百度也有类似的实现：disconf。

Diamond则是淘宝开源的一种分布式配置管理服务的实现。Diamond本质上是一个Java写的Web应用，其对外提供接口都是基于HTTP协议的，在阅读代码时可以从实现各个接口的controller入手。

分布式配置管理

分布式配置管理的本质基本上就是一种推送-订阅模式的运用。配置的应用方是订阅者，配置管理服务则是推送方。概括为下图：



其中，客户端包括管理人员publish数据到配置管理服务，可以理解为添加/更新数据；配置管理服务notify数据到订阅者，可以理解为推送。

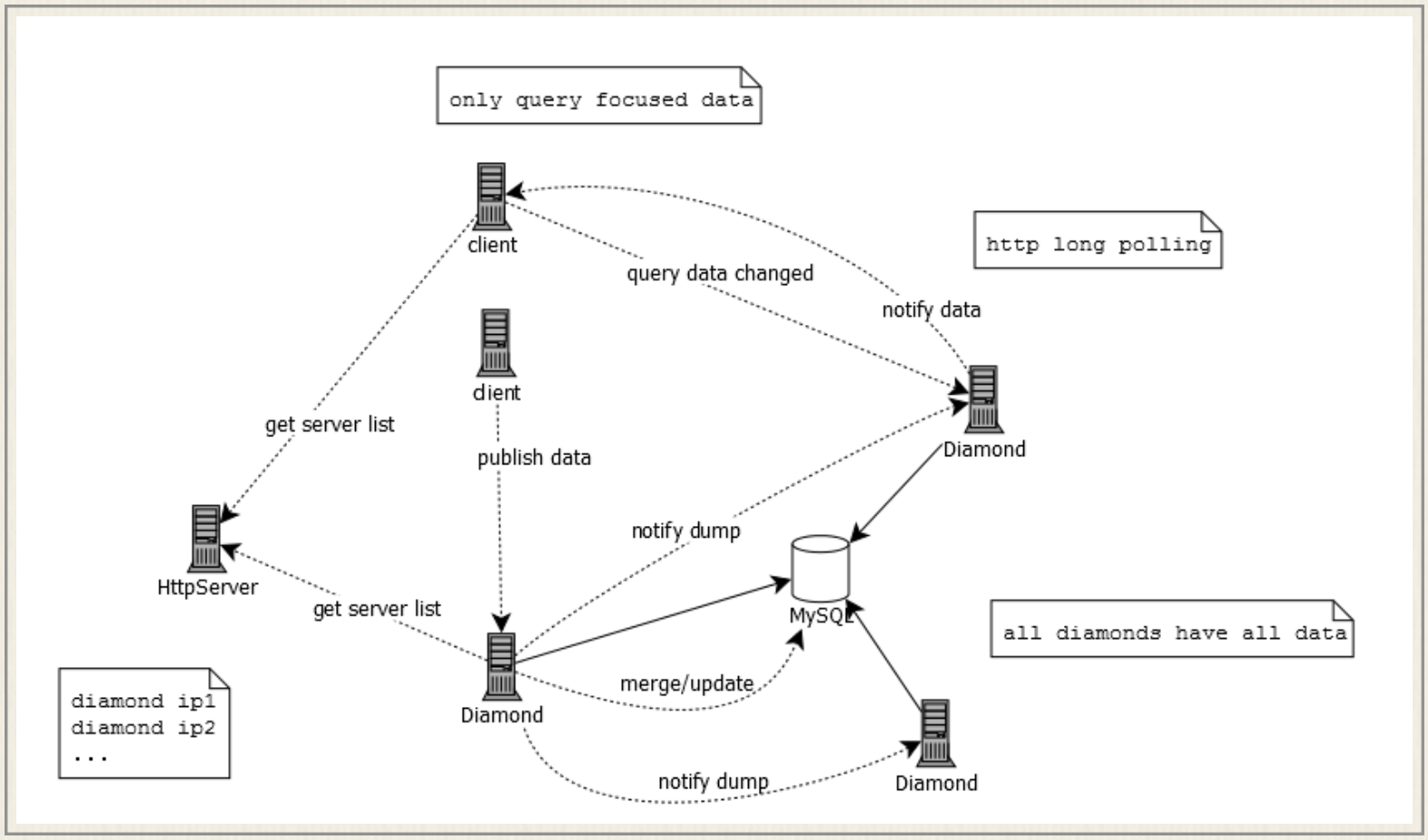
配置管理服务往往会封装一个客户端库，应用方则是基于该库与配置管理服务进行交互。在实际实现时，客户端库可能是主动拉取(pull)数据，但对于应用方而言，一般是一种事件通知方式。

Diamond中的数据是简单的key-value结构。应用方订阅数据则是基于key来订阅，未订阅的数据当然不会被推送。数据从类型上又划分为 聚合和非聚合。因为数据推送者可能很多，在整个分布式环境中，可能有多个推送者在推送相同key的数据，这些数据如果是聚合的，那么所有这些推送者推送的 数据会被合并在一起；反之如果是非聚合的，则会出现覆盖。

数据的来源可能是人工通过管理端录入，也可能是其他服务通过配置管理服务的推送接口自动录入。

架构及实现

Diamond服务是一个集群，是一个去除了单点的协作集群。如图：



图中可分为以下部分讲解:

服务之间同步

Diamond服务集群每一个实例都可以对外完整地提供服务，那么意味着每个实例上都有整个集群维护的数据。Diamond有两种方式保证这一点:

- 任何一个实例都有其他实例的地址；任何一个实例上的数据变更时，都会将改变的数据同步到mysql上，然后通知其他所有实例从mysql上进行一次数据拉取(DumpService::dump)，这个过程只拉取改变了的数据
- 任何一个实例启动后都会以较长的时间间隔（几小时），从mysql进行一次全量的数据拉取(DumpAllProcessor)

实现上为了一致性，通知其他实例实际上也包含自己。以服务器收到添加聚合数据为例，处理过程大致为:

DatumController::addDatum // /datum.do?method=addDatum

PersistService::addAggrConfigInfo

MergeDatumService::addMergeTask // 添加一个*MergeData-Task*，异步处理

MergeTaskProcessor::process

PersistService::insertOrUpdate

EventDispatcher.fireEvent(new ConfigDataChangeEvent // 派发一个*ConfigDataChangeEvent*事件

NotifyService::onEvent // 接收事件并处理

TaskManager::addTask(..., new NotifyTask // 由此，当数据发生变动，则最终创建了一个*NoticyTask*

*// NotifyTask*同样异步处理

NotifyTaskProcessor::process

foreach server in serverList // 包含自己

notifyToDump // 调用 */notify.do?method=notifyConfigInfo* 从*mysql* 更新变动的数据

虽然Diamond去除了单点问题，不过问题都下降到了mysql上。但由于其作为配置管理的定位，其数据量就mysql的应用而言算小的了，所以可以一定程度上保证整个服务的可用性。

数据一致性

由于Diamond服务器没有master，任何一个实例都可以读写数据，那么针对同一个key的数据则可能面临冲突。这里应该是通过mysql来 保证数据的一致性。每一次客户端请求写数据时，Diamond都将写请求投递给mysql，然后通知集群内所有Diamond实例（包括自己）从 mysql拉取数据。当然，拉取数据则可能不是每一次写入都能拉出来，也就是最终一致性。

Diamond中没有把数据放入内存，但会放到本地文件。对于客户端的读操作而言，则是直接返回本地文件里的数据。

服务实例列表

Diamond 服务实例列表是一份静态数据，直接将每个实例的地址存放在一个web server上。无论是Diamond服务还是客户端都从该web server上取出实例列表。

对于客户端而言，当其取出了该列表后，则是随机选择一个节点(ServerListManager.java)，以后的请求都会发往该节点。

数据同步

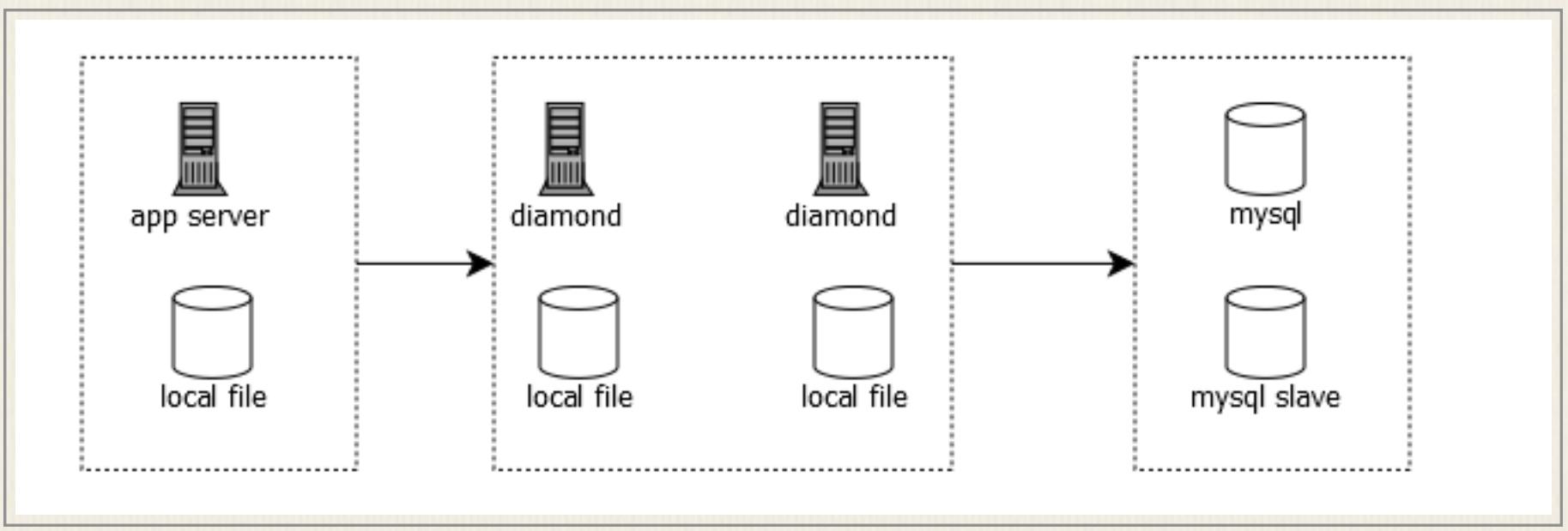
客户端库中以固定时间间隔从服务器拉取数据(ClientWorker::ClientWorker，ClientWorker::checkServerConfigInfo)。只有应用方关心的数据才可能被拉取。另外，为了数据推送的及时，Diamond还使用了一种long polling的技术，其实也是为了突破HTTP协议的局限性。如果整个服务是基于TCP的自定义协议，客户端与服务器保持长连接则没有这些问题。

数据的变更

Diamond中很多操作都会检查数据是否发生了变化。标识数据变化则是基于数据对应的MD5值来实现的。

容灾

在整个Diamond 系统中，几个角色为了提高容灾性，都有自己的缓存，概括为下图：



原文链接：http://codemacro.com/2014/10/12/diamond/?utm_source=tuicool

Objective-C之父Brad Cox访谈：我的编程之路

译者：伍昆

众所周知，Brad Cox与Tom Love一起创造了iOS/MAC平台的首选开发语言Objective-C。构建于C语言之上的Objective-C其理念来自于公认的第二个面向对 象的程序设计语言和第一个真正的集成开发环境的Smalltalk 。日前，原文作者Dave Dribin采访了Brad Cox，一起来了解下这位大神级人物，是如何看待他的编程之路。

Q：请您简单做个自我介绍以及与Objective-C的关系，谢谢。

Brad Cox：我 之前在ITT实验室工作，在那里结识了Tom，我们都有兴趣为人们带来高效的协作系统，但不同于邮件和实时通讯等应用；现在人们习惯称呼这类系统为 Groupware(群组软件)。我们发现在C语言上很难实现想法，当接触了Smalltalk以及面向对象的概念后，我想我们是时候要做出改变了，我们 要创造一个更好用的基础平台。初期版本是有关Sed和AWK的工具集合，后来发展为基于lex和yacc的语言。而今天它还可以作为汇编语言的生成工具。

Q：方括号可以说是Objective-C的特色，请问这是您的杰作吗?

Brad Cox：是的，其实我就想看看有什么还没使用过的，花括号有了，圆括号有了，最后就是方括号了，我希望人们使用时不会与C语言等类似的混淆。

Q：有什么功能您当初想添加的，但限于当时的技术和时间等制约因素，而最后没有完成?

Brad Cox：的确曾有过不少想法。例如Smalltalk中的闭包概念，垃圾回收机制，解释型语言等。这些想法后来通过不同形式被实现了，但是没有任何一个类在C语言中是绝对适合的。

今天这些的确都一一成为了现实。在Mac OS X 10.5中，给出了垃圾回收机制；在10.6中，也就是雪豹版本，在Objective-C中添加了闭包处理。

Brad Cox：是的，我一直是闭包的拥护者，但是要进入调用者的堆栈的确是需要一定的技巧。

Q：在静态和动态还一直争论不休的时候，Objective-C做了个跨界整合，动态的运行时环境和静态的编译。这是故意而为还是意外行为?

Brad Cox：事实上，很多功能是在我之后发布的。而当时我直接参与的，是在C的静态基础上简单地添加动态支持。而后来，静态支持是在这之后发展起来的。Objective-C的功能都是希望为大家带来非常轻量级的工具，这也是Objective-C一直追求的。

与纯静态的C++和Java相比，Objective-C 的动态类型支持的确非常好用。

Brad Cox：可能是多了一种轻巧的预设组件方式组合。

Q：苹果iPod和iPhone的销量超过了3千万，看到如此多的手持设备运行在Objective-C之上，那感觉是怎么样的?

Brad Cox：这感觉美妙极了。

Q：语言的设计是非常呆板的。过去20多年间出现Java、C#、Python及Ruby等都与Objective-C有很大不同。函数式语言似乎稍稍打破了沉闷的环境。对于语言设计的下一个重大事件或功能，您有什么看法呢?

Brad Cox：嗯，函数式语言现在变得热门了。我也曾尝试过，但是语法这个环节我就碰了壁。可能还欠点缘分。

Q：您觉得是语言的选择影响了最终软件质量，还是说全部的面向对象语言其实都大同小异?

Brad Cox：我想他们大体上都是差不多的。

Q：从您近期发布的消息和您的新作品<<Superdistribution>>一书中，不难看出您关注的焦点转移到了软件组件部分。请问能更多地讲述您现在的兴趣点吗?

Brad Cox：其实我没有转变焦点，请记得，为人们带来轻巧实用的功能是我创造Objective-C的初衷。对于我有关组件的关注，其实是我希望能找出有效帮助打造协作系统的办公自动化组件，所以说我并没有转变，只是希望帮助语言设计找到一个绕弯的方法到达相同的目的地。有关组件方向的更多介绍，请进入我的个人博客进行了解。

另外，我对OSGi模块化架构有非常大的期待。一旦流行起来，它将扮演非常重要的角色。虽然上手有一定难度，但是尽早学习是我的建议。

Q：可以分享更多有关OSGi的看法吗?

Brad Cox：好的。OSGi运行在Objective-C上可能不太合适，但是在其它的Java平台运作良好。一些主流的IDEs开发平台，如Netbeans和Eclipse，是一个能实现过渡到基于OSGi组件的中间平台。而随着技术的日渐发展和逐步成熟，将来或会成为人们日常工作的一部分。

Q：OSGi会替代jars吗？还是会与jars相结合?

Brad Cox：我倾向于后者。

Q：是metadata提取物?

Brad Cox：是的。一个OSGi bundle可以看成是一个metadata的jar提取物。因此也可以说是在JVM中的小型SOA(基于服务架构)服务。他们拥有自己的生命周期，能实现离线等待等处理。

Q：这样一来，能实现在线组件更新而不用把整个系统关闭?

Brad Cox：没错。一个典型的做法是在虚拟机中运行OSGi，然后等候它来完成升级过程。这需要花时间来熟悉。这个模型与Java中不断创建—销毁的过程不太一样。

译文链接：http://www.csdn.net/article/2014-10-08/2821983-Objective-C-BradCox

原文链接：http://www.mactech.com/articles/mactech/Vol.25/25.07/2507RoadtoCode-BradCoxInterview/index.html